A Small C Compiler
2nd Edition

DISK INCLUDED!

The quick and easy way to learn C, enjoy its power and flexibility, and discover the workings of a compiler. Disk includes a fully functional Small C compiler!



James E. Hendrix

M&T BOOKS

A Small C Compiler
2nd Edition
HENDRIX
DISK INCLUDED!
M&T BOOKS

A Small C Compiler

2nd Edition

A Small C Compiler

2nd Edition



James E. Hendrix

M&T BOOKS

M&T Books
A Division of M&T Publishing, Inc.
501 Galveston Drive
Redwood City, CA 94063

© 1990 by M&T Publishing, Inc.

Printed in the United States of America
First Edition published 1988

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval sytem, without prior written pemission from the Publisher. Contact the Publisher for information on foreign rights.

Limits of Liability and Disclaimer of Warranty
The Author and Publisher of this book have used their best efforts in preparing the book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness.

The Author and Publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The Author and Publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Library of Congress Cataloging in Publication Data

Hendrix, James E.
A Small C Compiler

Bibliography: p. 587
Includes index.
1. Small-C (Computer program language) 2. Compilers (Computer programs) I. Title.
QA76.73.S58H45 1988 005.13'3 88-9291
ISBN 0-934375-88-7
ISBN 1-55851-007-9 (disk)
ISBN 0-934375-97-6 (book & disk)

92 91 90 5 4 3 2 1

Trademarks:
CP/M is a registered trademark of Digital Research Inc.
UNIX is a registered trademark of AT&T Bell Laboratories.
Microsoft is a registered trademark of Microsoft Corporation.
MS-DOS is a registered trademark of Microsoft Corporation.
XENIX is a registered trademark of Microsoft Corporation.
IBM is a registered trademark of International Business Machines Corporation.
PC-DOS is a registered trademark of International Business Machines Corporation.
All products, names, and services are trademarks or registered trademarks of their respective companies.

Cover Design: Lauren Smith Design
Cover Illustration: Christine Mortensen

To

Ed and Billie
Wiley and Etta

Contents

PREFACE **5**

INTRODUCTION **9**

PART 1: THE SMALL C LANGUAGE **13**

1. Program Structure 19
2. Tokens 27
3. Constants 33
4. Variables 39
5. Pointers 49
6. Arrays 57
7. Initial Values 65
8. Functions 69
9. Expressions 89
10. Statements 111
11. Preprocessor Directives 125

PART 2: THE SMALL C COMPILER **131**

12. Library Functions 135
13. Efficiency Considerations 165
14. Compatibility with Full C 173
15. Executing Small C Programs 185
16. Compiling Small C Programs 191
17. Compiling the Compiler 199

PART 3: INSIDE THE SMALL C COMPILER **203**

18. The CALL Routines 209
19. Generated Code 215
20. Data Structures 247

21. The Back End 259
22. The Front End 283
23. The Main Function 293
24. High-Level Parsing 297
25. Low-Level Parsing 309
26. Expression Analysis 329
27. Optimizing 379
28. Further Development 395

APPENDICES

A. The ASCII Character Set 427
B. The 80x86-Family Processors 429
C. Small C Compiler Listings 437
D. Small C Library Listings 509
E. Small C Quick Reference Guide 561
F. Small C Error Messages 571
G. Changes from Small C 2.1 579
H. Index of Compiler Functions 583
I. Index of Library Functions 585

BIBLIOGRAPHY 587

INDEX 591

FIGURES

5-1. Small C Memory Model 52
8-1. The Stack Frame 83
15-1. Arguments Passed to Small C Programs 187
16-1. Compiling Small C Programs 192
17-1. Compiling the Compiler 199
P3-1. Organization of the Small C Compiler 206
18-1. Set-up When __SWITCH is Called 213
20-1. Format of the Macro Name Table 249

20-2. Format of the Switch Table 251
20-3. Format of the Symbol Table 253
20-4. Format of the While Queue 258
22-1. Major Front-End Functions 285
24-1. Primary High-Level Parsing Functions 299
25-1. Primary Low-Level Parsing Functions 309
25-2. Structure of IF and IF/ELSE Statements 318
25-3. Structure of a WHILE Statement 320
25-4. Structure of a DO Statement 322
25-5. Structure of a FOR Statement 322
25-6. Structure of a SWITCH Statement 324
26-1. Expression Analysis Functions 330
26-2. Is[] Arrays for i=j+k/5; 339
27-1. P-code Property Byte 381
B-1. 8086 CPU Architecture 430
B-2. 8086 Instruction Address 431
B-3. 8086 Stack Address 432
B-4. Four Component Data Address 434

TABLES

4-1. Variable Declarations 45
5-1. Pointer Declarations 50
6-1. Array Declarations 58
7-1. Permitted Object/Initializer Combinations 68
8-1. Function Declarations 73
9-1. Small C Operators 90
12-1. Standard File-Descriptor Assignments 137
12-2. Printf Examples 147
12-3. Small C Codes for Auxiliary Keystrokes 163
15-1. Redirecting Standard Input and Output Files 189
16-1. Invoking the Compiler 197
20-1. Data Structure Pointers 247

20-2. Defined Values for the Identity Field 254
20-3. Defined Values for the Type Field....254
20-4. Defined Values for the Class Field....255
21-1. Small C P-code Legend....268
21-2. Compiler Generated P-codes....269
21-3. Optimizer Generated P-codes....272
21-4. P-code Translation Strings 274
25-1. Statement Tokens, Parsers, and Lastst Values....311
26-1. Is[] Array Contents....340
26-2. Original and Optimized Relational Test P-codes 343
26-3. Contents of the Op[] Arrays....348
26-4. Values Returned by Primary()....374
27-1. Masks for Decoding P-code Property Bytes 382
27-2. Optimization Metacodes 385
B-1. 8086 Operand Addressing Modes 432

LISTINGS

1-1. Sample Small C Program 19
4-1. The Scope of Local Variables....43
5-1. Example of the Use of Pointers....56
6-1. Example of the Use of Arrays....63
8-1. Sample Recursive Function Call 86
8-2. String Comparison Function 87
17-1. Batch File to Compile the Compiler 200
17-2. Batch File to Recompile One Part of the Compiler 201
19-1. Code Generated by Constant Expressions 217
19-2. Code Generated by Global Objects....219
19-3. Code Generated by Global References 221
19-4. Code Generated by External Declarations 223
19-5. Code Generated by External References....223
19-6. Code Generated by Local Objects/References....225
19-7. Code Generated by Function Arguments/References 228

19-8. Code Generated by Direct Function Calls.....229
19-9. Code Generated by Indirect Function Calls.....230
19-10. Code Generated by the Logical NOT Operator.....231
19-11. Code Generated by the Increment Prefix.....231
19-12. Code Generated by the Increment Suffix.....231
19-13. Code Generated by the Indirection Operator.....232
19-14. Code Generated by the Address Operator.....233
19-15. Code Generated by Division and Modulo Operators....233
19-16. Code Generated by the Addition Operator.....234
19-17. Code Generated by the Equality Operator.....234
19-18. Code Generated by the Logical AND Operator.....236
19-19. Code Generated by Assignment Operators.....237
19-20. Code Generated by a Complex Expression.....237
19-21. Code Generated by an IF Statement.....238
19-22. Code Generated by an IF/ELSE Statement.....239
19-23. Code Generated by Non-Zero and Zero Tests.....239
19-24. Code Generated by a SWITCH Statement.....241
19-25. Code Generated by a WHILE Statement.....242
19-26. Code Generated by a FOR Statement.....243
19-27. Code Generated by a FOR Without Expressions.....244
19-28. Code Generated by a DO/WHILE Statement.....245
19-29. Code Generated by a GOTO Statement.....245
26-1. Pseudocode Representation of Down2().....352
26-2. Pseudocode Representation of Level13().....362
26-3. Pseudocode Representation of Level14().....367

Why This Book is for You

A Small C Compiler thoroughly documents every aspect of the popular Small C language and compiler, making Small C the only implementation of the C language with no secrets; everything is explained to the reader. The language itself is presented succinctly and in a way that makes learning C both easy and quick. And, since the diskette is included with the book, Small C is an economical introduction to the world of C programming for programmers who can wait for real numbers.

But for the truly curious, there is more. Ten chapters are devoted to explaining how the computer works, and all of the source code for the compiler and its function library is printed in the book for easy reference and is included on the diskette. Finally, one chapter discusses projects for further development of the compiler—projects which the ambitious reader can do using Small C to create new versions of itself. Thus, *A Small C Compiler* is an open ended path into Small C—a path which a reader may travel as far as curiosity takes him.

How to Order an Additional Disk

All the software listings in this book are available on disk (MS-DOS format) with full source code. The disk includes a full, working Small C Compiler, a run-time library, source code for all compiler and run-time library source files, and an executable assembler.

The disk price is $25.00. California residents must add the appropriate sales tax.

Order by sending a check, or credit card number and expiration date, to:



Small C Compiler Disk
M&T Books
501 Galveston Drive
Redwood City, CA 94063

Or, you may order by calling our toll-free number Monday through Friday, beween 8 A.M. and 5:00 P.M. Pacific Standard Time: 800/533-4372 (800/356-2002 in California). Ask for **Item #007-9**.

Preface

Since Ron Cain introduced the original Small C compiler in 1980, interest has grown steadily. Even with today's inexpensive full-featured C compilers, the popularity of Small C is increasing. The reasons are evident. Small C is a self-compiler, it comes with all of its source code (not just the library), it generates readable assembly code, and its single-pass algorithm lets users compile individual statements from the keyboard to the screen as a means of studying the compiler's behavior. Yet, despite its student orientation, it has the feel of a quality compiler and is easily capable of supporting serious development efforts in areas that do not require floating point numbers. And, finally, it is perhaps the most thoroughly documented compiler available to the public.

My first book on Small C, *The Small C Handbook,* was written for the CP/M version of the compiler. While CP/M still flourishes in Japan, and to a lesser extent in Europe, here in the United States it has long since given way to PC/MS-DOS. The compiler made the transition in 1985, but the documentation has taken a while to catch up. This is not without its advantages, however. Since its original transport to DOS, I have refined Small C repeatedly—especially the code generating and optimizing sections that were rewritten twice—so that the current version is neater, far easier to understand, and generates much better code. The result is that the compiler described here is vastly improved over the one that would have been documented in 1985.

This book is more comprehensive than its predecessor. Whereas the *Handbook* was just that—a handbook for using the compiler, this book goes on to treat the internal operation of the compiler. Following the advice of reviewers, I have fleshed out the text with better explanations and more examples. A chapter of ideas for further development of the compiler has been included as a basis for student projects.

Feeling that the original book tried to cover too much territory, I split the subject matter into two volumes—this one at the C language level and another at the assembly language level. The second volume will deal with Small C's companion assembler in the same way this one treats Small C. It will introduce assembly language programming, linking, and the use of object libraries.

The general plan of the present book comprises three parts. Part 1 is an introduction to the Small C language—an integer/character subset of the C language set forth by Kernighan and Ritchie in *The C Programming Language* [10]. Since Small C is a faithful subset of C, this material may serve as an introduction to the full language. The concepts and syntax are the same, so moving up to Full C from Small C is just a matter of learning those features of C which Small C does not support. The point is that one can begin learning C here, without having to deal with the more advanced features of the language.

Part 2 describes the use of the compiler. The library of standard functions is documented from a programmer's point of view. Efficiency considerations are discussed, as is the matter of compatibility with Full C. The reader is also taught how to invoke the Small C compiler and how to use it to compile new versions of itself.

Part 3 gives a play-by-play description of the compiler's operation. Both top-down and bottom-up approaches are taken, with everything coming together in the middle where the parsing is done. Because it is not germane to the subject at hand, and because I wanted to keep the size of the book down, I chose not to cover the library's theory of operation. Nevertheless, a complete listing can be found in Appendix D, and since it is basically a collection of trivial routines, it makes easy reading without further commentary.

Appendices are provided which survey the 80x86 processors, list the compiler and library, summarize Small C syntax, explain error messages, and document differences between this version of the compiler and its predecessor. In addition, there are two appendices which index the functions of the compiler and the library; they give each function's source file, its page in the text, and its page in the source listings.

I wish to express sincere appreciation to those who encouraged and assisted me in this work. That necessarily begins with Ron Cain, who wrote the original Small C compiler and entrusted its future to me. For his encouragement in writing the *Handbook,* I remain grateful to Marlin Ouverson. And, for reviewing that manuscript, I am indebted to George Boswell, Hal Fulton, and Jim Wahlman. For his invaluable assistance in reviewing the present manuscript, I wish to thank Tobin Maginnis. To all of those who, over the years, have shared with me their fixes, enhancements, and suggestions, I would like to say thanks for sharing in the evolution of Small C. Special thanks to Ellen Ablow, Sally Brenton, Michelle Hudun, Dave Rosenthal, Ann Roskey, and others behind the scenes at M&T Publishing, for their hard work and cooperation in this project. Finally, to my wife Glenda, who has patiently seen me through yet another Small C book, I express my sincerest appreciation.

James E. Hendrix

Introduction

The C programming language was developed at Bell Laboratories in the early seventies by Dennis Ritchie, who based his work on Ken Thompson's B language. C was designed to conveniently manipulate the same kinds of objects known to computer processors—bits, bytes, words, and addresses. For that reason, and because it is a structured language, it has become the language of choice for systems programming on minicomputers and microcomputers. It was originally developed for implementing the UNIX operating system, and today PC/MS-DOS is largely written in C.

C has other good applications too. It is well-suited to text processing, engineering, and simulation problems. Although other languages have specific features which, in many cases, better equip them for particular tasks (e.g., the complex numbers of FORTRAN, the matrix operations of PL/I, and the sort verb, report writer, and edited moves of COBOL), C is nevertheless a very popular language for a wide range of applications, and for good reason—programmers like it.

Users of C typically cite the following reasons for their satisfaction: (1) C programs tend to be more portable; (2) C has a rich set of expression operators, making it unnecessary to resort to assembly language except in rare cases; (3) C programs are compact, but are not necessarily cryptic; (4) C compilers usually generate efficient object code; and (5) C is a relaxed language, without unnecessarily awkward syntax.

For a description of the complete C language, I refer you to the original book on the subject, *The C Programming Language,* by Brian W. Kernighan and Dennis M. Ritchie [10]. Although numerous other books have appeared, and a standard dialect of C is emerging, this remains the primary non-vendor source on the language.

A SMALL C COMPILER

In May of 1980, *Dr. Dobb's Journal* ran an article entitled "A Small C Compiler for the 8080s" in which Ron Cain presented a small compiler for a subset of the C language. The most interesting feature of the compiler, besides its small size, was the language in which it was written—the one it compiled. It was a self-compiler! (Although this is commonplace today, it was a fairly novel idea at the time.) With a simple, one-pass algorithm, his compiler generated assembly language for the 8080 processor. Being small, however, it had its limitations. It recognized only characters, integers, and single-dimension arrays of either type. The only loop-controlling device was the **while** statement. There were no Boolean operators, so the bitwise logical operators **&** (AND) and | (OR) were used instead. Even with these limitations, it was a very capable language and a delight to use, especially compared to assembly language.

Recognizing the need for improvements, Ron encouraged me to produce a second version, and in December of 1982 it also appeared in *Dr. Dobb's Journal.* The new compiler augmented Small C with (1) code optimizing, (2) data initializing, (3) conditional compiling, (4) the **extern** storage class, (5) the **for, do/while, switch,** and **goto** statements, (6) combination assignment operators, (7) Boolean operators, (8) the one's complement operator, (9) block local variables, and various other features. Then in 1984, Ernest Payne and I developed and published a CP/M-compatible, run-time library for the compiler. It consisted of over 80 functions and included most of those in the UNIX C Standard I/O Library—the ones that pertained to the CP/M environment. This became version 2.1 and the subject of *The Small C Handbook.*

Within a year, Russ Nelson, of Clarkson College, had this compiler running under MS-DOS. And, through an agreement with them, I was able to base my own 8086 implementation on his work. Although I revised the compiler extensively, his primary contribution—the use of p-codes—has remained. The run-time library was thoroughly reworked, adapting the input/output functions to the DOS file handle facility. This is the DOS compiler that *Dr. Dobb's Journal* distributed as version 2.1.

On two occasions since, I have revisited the compiler to overhaul its code generator. First, I replaced the long string of **if...else...** statements that translated

p-codes to assembly code, with a huge **switch** statement; then I replaced that with an array of pointers to assembly code strings. Translating the p-codes became just a matter of subscripting the array with the p-codes themselves.

Finally, while sprucing up the compiler for this book, I rewrote the code optimizer from scratch. Whereas before it consisted of a string of if...else... statements that looked for specific sequences of p-codes and replaced them with others, it has now been generalized, reduced in size, and made to do more optimizing. The result: Small C now generates code that is respectable compared to professional compilers. Another advantage is that it is now very easy to understand what the optimizer does and to add to its repertoire of tricks. (See Appendix G for a complete list of differences between versions 2.1 and 2.2.)

This version of Small C (2.2) is the subject of this book. It remains a subset compiler, but is now better organized and much more efficient. I have resisted the urge to develop Small C into a full C compiler for a number of reasons. First, it would take a lot of work. But more importantly, it would move Small C out of its niche as a student compiler by obscuring its logic with additional complications. Being incomplete, Small C has plenty of room for improvement. Students can experiment with it, adding missing features and improving its algorithms. Chapter 28 lists many possibilities for further developing the compiler.

As you read along, I trust that you will enjoy learning a really neat language and experience the satisfaction of learning the mysteries of compiler operation.

Part 1

The Small C Language

PART 1

The Small C Language

The following chapters introduce the Small C language in a natural order. Since each chapter deals with a narrow aspect of the language, however, they may also be read independently for reference purposes.

Conventions Followed in the Text

Although most of the concepts in the following chapters pertain to both Small C and Full C, some relate only to Small C or only to Full C. So, to avoid confusion, I refer to these languages in a systematic way. The term C, with no adjective, refers to both Small C and Full C. Statements about Small C apply only to Small C, and statements about Full C pertain only to the complete language.

Following customary practice, functions are distinguished from other things by suffixing their names with a set of matching parentheses. Because it follows C syntax, programmers instinctively think "it's a function" when they see this. Similarly, array names are followed by matching square brackets.

Occasionally you will see the ellipsis (...) used as an abbreviation for anything that might appear at some point. An example is **if...else...** which appears a few paragraphs below. Note that the ellipsis assumes special meaning in syntax definitions.

Syntax definitions employ the following devices:

1. Generic terms are italicized and begin with a capital letter. They specify the kind of item that must or may appear at some point in the syntax. Frequently, two terms are combined to identify a single item.
2. Symbols and special characters in boldface are required by the syntax. They must appear exactly as shown.

3. The term *string* refers to a contiguous string of items.

4. The term *list* refers to a series of items separated by commas and (possibly) white space.

5. An ellipsis (...) means that entities of the same type may be repeated any number of times.

6. A question mark at the end of a term identifies it as optional; it may or may not be needed.

The 80x86 Family of Processors

Since an understanding of any compiled language requires some knowledge of the way data is represented in the processor that interprets the compiled programs, you will find numerous references to the 8086 processor. The Intel 80x86 was the first of a family of 16-bit processors upon which IBM's personal computers, and compatible machines, are based. See Appendix B for a survey of the architecture of these processors.

A Sample Program

Before proceeding, let's get a feel for the C language by surveying a small program. Listing 1-1 is a program, called Words, which takes each word from an input file and places it on a line by itself in an output file. A word, in this case, is any contiguous string of printable characters.

The first line of the program is a comment, giving the name of the program and a brief description of its function. The second line instructs the compiler to include text from the file **stdio.h.** The included text appears to the compiler exactly as though it had been written in place of the **#include** directive. The third and fourth lines define the symbols **INSIDE** and **OUTSIDE** to stand for one and zero, respectively. A preprocessor built into the compiler scans each line, replacing all such symbols with the values they represent. Such symbols are often called *macros,* since they stand for substitution text that may be larger and more complex than its name.

The next two lines define variables, a character named **ch,** and an integer named **where** that is given the initial value zero (represented by **OUTSIDE**).

The procedural part of the program consists of three functions: **main(), white(), and black().** Execution begins in **main(),** which contains calls to **white()** and **black().** The **while** statement in **main()** controls repeated execution of the

```
if...else...
```

statement enclosed in braces. With each repetition, it calls **getchar()** to obtain the next character from the input file. (Although **getchar()** is not defined in this program, it is nevertheless available for use because it exists in a library of functions that can be linked with the program.)

The character obtained is assigned to the character variable **ch.** If it is not equal (!=) to the value represented by **EOF** (defined in **stdio.h**), then the **if** statement is performed; otherwise, control passes through the end of **main()** and back to the operating system.

With each iteration, the current character is checked to see if it is a white character—space (' '), newline ('\n'), or tab ('\t'). If so, **white()** is called. **White()** then checks to see if the previous character was within a word (**where** equals **INSIDE**). If so, the current character must be the first white character following a word, therefore **putchar()** (another library function) is called to write a special newline character to the output file to terminate the current line and start another one. It then sets **where** to **OUTSIDE** so that no more new lines will be written for that word. When the next black (printable) character is found, a call is made to **black()** which writes the character to the output file and sets the variable **where** to **INSIDE,** indicating that the most recent character was within a word.

As this program executes, it has the effect of squeezing all continuous runs of white characters into a single occurrence of the newline character that has the effect of a carriage return, line feed sequence.

Although this quick tour through a C program leaves many questions unanswered, it illustrates the general form of C programs, how variables and functions are defined, and how control flows through a programs. The following chapters fully explain these concepts.

CHAPTER 1

Program Structure

C is a free field language. There is no significance associated with any particular position within a line, and both multi-statement lines and multi-line statements are allowed. The only exceptions to this rule are the preprocessor directives (Chapter 11), each written on a line by itself, beginning with **#.**

Statements and declarations may be split between lines at any point except within keywords, names, and multi-character operators. That is, *tokens* (the smallest units of the syntax) must be wholly contained in a line. Another way to think of this is that C interprets line breaks—blanks and tabs—as *white space.*

Overall Structure

As Listing 1-1 shows, C programs have a simple structure. A program is essentially a sequence of *comments* (line 1), *preprocessor directives* (lines 2-4), and *global declarations* (beginning with lines 5, 6, 7, 14, and 18). Notice that the procedural parts of a program—the *functions*—are considered to be declarations.

```
1  /* words - put every word on a line by itself */
2  #include <stdio.h>
3  #define INSIDE  1
4  #define OUTSIDE 0
5  char ch;
6  int where = OUTSIDE;
7  main() {
8    while((ch = getchar()) != EOF) {
9      if((ch == ' ') || (ch == '\n') || (ch == '\t'))
10           white();
11      else black();
12      }
13    }
14  white() {
```

```
15    if(where == INSIDE) putchar('\n');
16    where = OUTSIDE;
17    }
18  black() {
19    putchar(ch);
20    where = INSIDE;
21    }
```

Listing 1-1. Sample Small C Program

Comments

Comments are used to clarify the logic of programs and document their operation. They are of no use to the compiler, so the preprocessor squeezes them out of a program before the compiler sees it. Comments are delimited on the left by the /* sequence and on the right by the next */ sequence. There is no limit to the length of comments and they may continue over any number of lines. We can place them anywhere in a program, except within tokens; that is, they are legal anywhere white space is allowed.

The preprocessor is sensitive to the comment delimiters only when they do not occur within quotes or apostrophes. So, we may freely use these character sequences in strings and character constants without fear of confusing the compiler. (The idea of a two-character constant may seem strange, but C accepts them without complaint.)

While some C compilers support "comment nesting" as an option (it is not a standard feature of C), Small C will definitely be confused by comments within comments. This makes it difficult to comment out of programs sections of logic that are themselves commented. This lack of comment nesting is an unfortunate characteristic of C.

Preprocessor Directives

Although not a part of the C language proper, I included preprocessor directives in the overall structure of C programs (see above) because they do actually appear in the source file together with the "pure" C code. We have already seen two examples of these directives in Listing 1-1 (lines 2–4); those and others will

be covered in detail in Chapter 11.

However, a few general remarks are in order. First, all C compilers include a preprocessor which prepares the source code for assimilation by the compiler. Small C is unusual in that the preprocessor is integrated into the compiler itself as a low-level input routine. This saves one pass of writing and reading the source code. It also means that when we probe the compiler with keyboard input, we may enter preprocessor directives along with regular C code.

The Small C preprocessor supports three capabilities: (1) the inclusion of source code from other files; (2) the definition of symbols that stand for arbitrary substitution text; and (3) the conditional compiling of optional program code.

Each preprocessor directive is written on a line by itself beginning with a # character. The preprocessor lines are eliminated from the program before it goes on the compiler.

Global Declarations

Objects that may be declared in Small C programs include:

1. integer variables
2. character variables
3. arrays of integers or characters
4. pointers to integers or characters
5. functions

Examples of these were seen in Listing 1-1.

Full C provides other kinds of declarations. In Full C, for instance, we can declare arrays of *pointers, structures* (collections of objects into records), and *unions* (redefinitions of storage). While Small C does not support these, it manages pretty well without them. For example, since pointers are the same size as integers, an array of integers has no problem holding pointers. And, while this leads to some inefficiencies and programming practices that are generally frowned upon, the language is not crippled by its inability to declare arrays of pointers. Also, the lack of structures and unions does not impose any real limitation because they provide no capabilities that cannot be done in other ways.

Since there is much to be said about function declarations, they are treated more thoroughly below. First, however, we need to distinguish between two similar terms: declarations and definitions.

Declarations and Definitions

The terms *declaration* and *definition* may seem synonymous, but in the C language they imply different ideas. Declarations only tell the compiler what it needs to know about objects; whereas definitions also reserve space in memory for them. We can declare something without defining it, but we cannot define something without declaring it.

An object may be said to *exist* in the file in which it is defined, since compiling the file yields a module containing the object. On the other hand, an object may be declared within a file in which it does not exist. Declarations of this type are preceded by the keyword **extern.** Thus,

```
int i;
```

defines an integer that exists in the present module; whereas,

```
extern int ei;
```

only declares an integer to exist in another, separately compiled, module. The *linker* brings these modules together and connects inter-module references by name.

The compiler knows everything about **extern** objects except where they are. The linker is responsible for resolving that discrepancy. The compiler simply tells the assembler that the objects are, in fact, external. The assembler, in turn, makes this known to the linker.

Function Declarations

A *function* is simply a subroutine—a common piece of logic that is *called* from various points in a program, and returns control to the caller when its work is done. Other languages distinguish between *functions* and *procedures,* the former being invoked as a term, returning a value, in expressions and the latter

being called directly by means of *call* statements. C eliminates the distinction by accepting a bare expression as a statement. Furthermore, since a function call may comprise an entire expression, functions can be called without the aid of a special statement.

While other block-structured languages support the nesting of procedural declarations, C does not. In C, all function declarations must occur at the *global* level—outside of other function declarations. This greatly simplifies the structure of C programs and makes them easier to understand.

A function declaration consists of two parts: a *declarator* and a *body*. The declarator states the name of the function and (for use within the function) the names of arguments passed to it. In our sample program, no arguments were passed to the functions that were called, so each of the three function declarators specified a null argument list (empty parentheses). The parentheses are required even when there are no arguments.

The body of a function consists of argument declarations followed by a statement that performs the work. The argument declarations indicate to the function the types of the arguments listed in the declarator. Each argument must be declared, and only arguments may be declared at this point. It is the responsibility of the programmer to ensure that the correct types of arguments are passed to the function when it is called. Chapter 28 describes how modern C compilers assume this responsibility and suggests how Small C can be modified to do the same.

The reference above to a statement (singular) may be surprising. After all, Listing 1-1 plainly shows several statements in each function. The reference is correct, however, since the collected statements following the argument declarations of a function comprise a single *compound statement*.

Compound Statements

A *compound statement* (or *block*) is a sequence of statements, enclosed by braces, that stands in place of a single statement. Simple and compound statements are completely interchangeable as far as the syntax of the C language is concerned. Therefore, the statements that comprise a compound statement may

themselves be compound; that is, blocks can be nested. Thus, it is legal to write:

```
{ i=5; { j=6; k=7; } }
```

in which **j=6**; and **k=7**; comprise a block which, together with **i=5**;, comprises a larger block. Of course this offers no advantage over simply writing the three statements in sequence. As we shall see, however, C does not provide for sequences of statements except in blocks. So, if a sequence of statements is to be controlled by some condition, we would need to write something like:

```
if (i < 5) { i=5; j=6; k=7; }
```

instead of:

```
if (i < 5) i=5; j=6; k=7;
```

In the latter case, only **i=5**; is controlled by the condition **i<5.** The other assignments happen regardless of the value of **i**.

Flow of Control

When a function receives control, execution begins with its first statement. If control reaches the end of the body—the closing brace—it then returns to the point from which the function was called and continues onward from there.

C programs always begin execution with an ordinary call to a function named **main().** Consequently, there must be a **main()** function somewhere in the program. A return from that function (for that call) transfers control back to the operating system. Nothing prevents a program from calling **main()** itself, however. When that occurs, the return from **main()** transfers control back to the point of the call. Control returns to the operating system from **main()** only when the instance of **main()** called by the operating system returns control.

External and Global Variables

Variables declared outside of a function, like **ch** and **where** in Listing 1-1, are properly called *external* variables because they are defined outside of any function. While this is the standard term for these variables, it is confusing because there is another class of external variable—one that exists in a separately compiled source file. So there is a need to distinguish between variables that are merely external to functions and those that are also external to the present source file. To avoid this confusion, I have chosen to refer to external variables in the present source file as *globals* since they are known to all of the functions of the program. By contrast, *locals* are known only to the functions (more properly, blocks) in which they are declared. Hereafter, the word *external* is reserved for variables that exist in other, separately compiled, source files; that is, those declared with the keyword **extern.**

Local Variables

As we shall see later, variables can be declared within a compound statement. We call these *local* variables since they are known only to the block in which they appear, and to subordinate blocks. For example, we could write:

```
if (i < 5) { int x; x=j; j=k; k=x; }
```

if we wanted to swap **j** and **k** when **i < 5.** Notice that the local variable **x** is declared within the compound statement. Unlike globals, which are said to be *static,* locals are created dynamically when their block is entered, and they cease to exist when control leaves the block. Furthermore, local names supersede the names of globals and other locals declared at higher levels of nesting. Therefore, locals may be used freely without regard to the names of "outlying" variables; clashes cannot occur.

Source Files

As we noted above, C programs may consist of source code in more than one file. One method of bringing the parts together is to use the **#include** preprocessor directive which we saw in Listing 1-1. Another is to compile the

source files separately, then combine the separate object files as the program is being linked with library modules. This is very convenient since the linker must be used anyway. It permits us to break large programs into parts so that the whole program need not be recompiled with every change that is made. As we shall see, the Small C compiler consists of four parts, which are brought together in this way.

Although (truly) external variables must be declared as such with the keyword **extern,** the Small C compiler assumes that undeclared functions are external. In either case, it generates an **EXTRN** directive telling the assembler to inform the linker to make the necessary connection. Furthermore, each global declaration generates a **PUBLIC** directive which ensures that it can be seen from other modules.

Summary

In brief, a Small C program comprises one or more source files. Two mechanisms are provided for bringing together the parts. The text from one file may be included into another file by means of the **#include** directive (Chapter 11); or the parts may be compiled and assembled separately, then linked together.

Each source file consists of comments, preprocessor directives, and global declarations for variables, arrays, pointers, and functions. Each function in turn has a declarator and a body. The declarator names the function and gives local names to its arguments. The body declares the types of the arguments and specifies an algorithm in a block of local declarations, executable statements, and other blocks. And these blocks in turn have their own local declarations, statements, blocks, and so on.

Programs begin execution in a function called **main(),** and other functions receive control only when they are specifically called. A function is called by writing its name followed by a (possibly null) list of arguments enclosed in parentheses.

These concepts are just the foundation for the things that follow. Read on for the good stuff.

CHAPTER 2

Tokens

The smallest elements in the syntax of a language are called *tokens*. A token may be a single character, or a sequence of characters that form a single item. The first order of business for a compiler is to recognize the individual tokens from which the program is built. It then tries to recognize patterns of tokens as language constructs. Finally, it generates code to perform the activities which the language constructs specify. Since tokens are the building blocks of programs, we begin our study of the Small C language by defining its tokens.

ASCII Character Set

First we look at the character set from which tokens are constructed. Small C uses the ASCII character set. As Appendix A indicates, there are 128 defined characters in the ASCII set. The first 32 (values 0–31) and the last one (127) are classified as *control characters*. Among them are the horizontal tab (9), the carriage return (13), and the line feed (10), which are commonly found in C programs. These characters, and the space (32) character, are often called white space characters since they form the gaps which usually separate tokens.

Codes 32-126 include the "normal" characters; these include the *space* (32), the *numeric digits* (48-57), the *uppercase alphabetics* (65-90), the *lowercase alphabetics* (97-122), and the *special characters* (all other values). From these are built the tokens of the Small C language.

Constants

Numeric constants consist of an uninterrupted sequence of digits which is delimited by white space or special characters (operators or punctuation, respectively). Since only integers are known to Small C, the period (.) cannot appear in numeric constants. They may be written with a leading plus or minus sign, how-

ever. For more about numeric constants, see Chapter 3.

Character constants are written by enclosing an ASCII character in apostrophes. We write **'a'** for a constant with the ASCII value of the lowercase **a** (97). There are variations on this idea, as we shall see in Chapter 3.

String constants are written as a sequence of ASCII characters bounded by quotation marks. Thus, **"abc"** describes a string of characters containing the first three letters of the alphabet in lowercase. (See Chapter 3 for more about string constants.)

Keywords

Keywords (sometimes called *reserved words*), are the tokens that look like words or abbreviations and serve to distinguish between the different language constructs. Examples are **int** in:

```
int i, j, k;
```

and **while** in:

```
while (i < 5) x[i++] = 0;
```

Perhaps the first thing noticed in Listing 1-1 was that, for the most part, the program is written in lowercase letters. In fact, all keywords in the C language are written in lowercase.

Names

Names (also called *identifiers* or *symbols*) are used to identify specific variables, functions, and macros. Small C names may be any length, however, only the first eight characters have significance. Trailing characters are ignored. Thus, the names **nameindex1** and **nameindex2** are both seen by the compiler as **nameinde.** This limit of eight characters, while common, is not universal among C compilers.

Names must begin with a letter, and the remaining characters must be either letters or digits. The underscore character (_) counts as a letter, however. Thus,

the name **_abc** is perfectly legal, as is **a_b_c.**

Names may be written with both uppercase and lowercase letters, which are equivalent. It is customary, however, to generally use lowercase except for macro symbols. The practice of naming macros in uppercase calls attention to the fact that they are not variable names but defined symbols. To improve readability, one common practice is to capitalize the first letter of each term that goes into a name. **GetTwo,** for instance, reads better than **gettwo.**

Every global name defined to the Small C compiler generates an assembly language label of the same name, but preceded by an underscore. The purpose of the underscore is to avoid clashes with the assembler's reserved words. As you study the Small C library (Appendix D) you will notice that global variables and some functions are named with leading underscores. This common practice is to avoid clashing with names a programmer might choose. So as a matter of practice, we should not ordinarily name globals with leading underscores.

Since the compiler adds its own underscore, names written with a leading underscore appear in the assembly file with two leading underscores.

Locals cannot clash with assembler-reserved words or library globals. This is because locals are allocated on the stack and are referenced relative to the stack frame instead of by name.

Punctuation

Punctuation in C is done with semicolons, colons, commas, apostrophes, quotation marks, braces, brackets, and parentheses. Semicolons are primarily used as statement terminators. One is placed at the end of every simple statement. As illustrated by:

```
{ x = j; j = k; k = x; }
```

Even the last statement in a block requires a semicolon.

Preprocessor directives are an exception since they are not part of the C language proper, and each one exists in a line by itself. Semicolons also separate the three expressions in a **for** statement (Chapter 10), as illustrated by

```
for (i = 0; i < 10; i = i + 1) x[i] = 0;
```

Since C has a **goto** statement, there must be a way of designating the destination address for a jump. This is done by writing an ordinary name followed by a colon. Such a name is called a **label.** An example is:

```
loop:
...
goto loop;
```

Colons also terminate **case** and **default** prefixes which appear in **switch** statements (Chapter 10). Consider:

```
switch (var) {
   case  3: putchar('x');
   case  2: putchar('x');
   case  1: putchar('x');
   default: putchar(' ');
}
```

for example. These prefixes may be thought of as special labels since they are, in fact, targets for a transfer of control.

Commas separate items that appear in lists. Thus, three integers may be declared by:

```
int i, j, k;
```

Or, a function requiring four arguments might be called with the statement:

```
func (arg1, arg2, arg3, arg4);
```

Commas are also used to separate lists of expressions. Sometimes it adds clarity to a program if related variables are modified at the same place. For example:

```
while (++i, -k) abc ();
```

The value of a list of expressions is always the value of the last expression in the list.

Square brackets enclose array dimensions (in declarations) and subscripts (in expressions). Thus,

```
char string[80];
```

declares a character array named **string** consisting of 80 characters numbered from 0 through 79, and

```
ch = string[4];
```

assigns the fifth character of that array to the variable **ch.**

As we saw in Listing 1-1, parentheses enclose argument lists which are associated with function declarations and calls. They are required even if there are no arguments.

As with all programming languages, C uses parentheses to control the order in which expressions are evaluated. Thus, **(6+2)/2** yields 4, whereas **6+2/2** yields 7.

The backslash character (\) may be used in character and string constants as an escape character. It's presence gives some character(s) special meaning.

Operators

As Table 9-1 illustrates, numerous special characters are used as expression operators. They specify every sort of operation that can be performed on operands. There are operators for:

1. assignments
2. mathematical operations
3. relational comparisons
4. Boolean operations
5. bitwise operations
6. shifting values

7. calling functions
8. subscripting
9. obtaining the size of an object
10. obtaining the address of an object
11. referencing an object through its address
12. choosing between alternate subexpressions

Since there are so many operators, in many cases it is necessary to form operators from two or more characters. Thus <= and <<= are each operators. Small C requires that such operators be written without white space or comments between the characters.

CHAPTER 3

Constants

Small C recognizes three types of constants: *numeric, character,* and *string.* Furthermore, numeric constants may be written in three bases: *decimal, octal,* and *hexadecimal.* Each type of constant is discussed below.

Decimal Constants

Ordinary *decimal* constants are written as a sequence of decimal digits, possibly preceded by a plus or minus sign. The minus sign gives the constant a negative value, whereas the absence of a minus sign makes it positive. The plus sign is optional for positive values. The normal range of decimal constants is -32768 through +32767.

However, Small C allows us to write constants in the range 32768 through 65535. Former versions of the compiler would interpret these as their negative equivalents -32768 through -1—that is, the negative values with the same bit patterns. This is an unexpected interpretation, however, and it is not consistent with Full C compilers which preserve the value of large positive constants by converting them to long integers.

The current version of Small C takes these large positive constants as unsigned values. While this preserves their value and yields expected results, there is one caveat. In order to produce reasonable results, operations performed on unsigned values are necessarily unsigned operations. For most expression operators there is no difference between the signed and unsigned operations, but for the following operations there is a difference:

* multiplication
/ division
% modulo (remainder)

< less than
<= less than or equal
> greater than
>= greater than or equal

Also, when an unsigned operand is operated on (unary or binary operation), the result is considered to be unsigned. So, in complex expressions, we may get some unexpected unsigned operations.

There is one final caution about large positive values. Although Small C treats them as unsigned values, they nevertheless appear in the output file as negative values. But don't panic, this is really okay. Recall that each unsigned value over 32767 has the high-order bit set and so has an equivalent negative value with the same bit pattern. When these negative values pass through the assembler, they produce the same binary patterns as their unsigned equivalents. Consequently, the end result is the same.

With earlier versions of the Small C compiler, if we wanted an unsigned comparison we had to make sure that one of the operands was thought by the compiler to be an address. This could be done by falsely declaring an integer to be a character pointer (Chapter 5). And, in fact, this unorthodox practice has been used to good effect in many Small C programs. With the current compiler, however, we can write exactly what we intend by using the keyword **unsigned.**

As was implied in the previous discussion, decimal constants are reduced to their two's complement or unsigned binary equivalent and stored in 16-bit words. Some examples of legitimate decimal constants are **0, 12345, -1024,** and **+256.**

Octal Constants

If a sequence of digits begins with a leading **0** (zero) it is taken as an *octal* value. In this case the word *digits* refers only to the octal digits (0 through 7). As with decimal constants, octal constants are converted to their binary equivalent in 16-bit words. Octal constants may, therefore, range from 0 through 0177777. Here, as with decimal constants, we must realize that large values (100000 and higher) will be treated by the compiler as unsigned values. Some examples of

legitimate octal constants are **010, 01234,** and **077777.** Notice that the octal values 0 through 07 are equivalent to the decimal values 0 through 7.

The old CP/M version of Small C did not recognize octal constants. It took constants with leading zeroes as decimal values. Therefore, when converting a program written for that compiler we should strip leading zeroes from its numeric constants.

Hexadecimal Constants

If a sequence of digits begins with **0x** or **0X,** then it is taken as a *hexadecimal* value. In this case the word *digits* refers to hexadecimal digits (0 through F). The lowercase letters *a* through *f* are acceptable. As with decimal constants, hexadecimal constants are converted to their binary equivalent in 16-bit words. Hexadecimal constants may range from 0 through ffff. Here, as with decimal constants, we must realize that large values (8000 and higher) will be treated by the compiler as unsigned values. Some examples of legitimate hexadecimal constants are **0x0, 0x1234,** and **0xffff.**

Character Constants

Character constants consist of one or two characters surrounded by apostrophes. It may seem odd that a character constant could have two characters in it, but it makes sense when we consider that, like numeric constants, character constants become 16-bit words (integer-sized objects), with room for two characters.

The constant '**B**', for instance, produces 0042 hex (the ASCII value for an uppercase **B**). And the constant '**AB**' produces 4142 hex, which is simply the two characters **A** and **B** in the high-order and low-order bytes, respectively. Some compilers do not support multi-character constants because they present a portability problem when the byte order of the CPUs differ.

Character constants are always treated as signed integers. Therefore, unlike the large numeric constants mentioned above, character constants receive unsigned operations only when they are combined with unsigned operands. In either case, single-character constants are always positive because the high-order bit is always zero.

We should realize that the same is not always true of character variables. Unless a character variable is specifically declared to be unsigned, its high-order bit will be taken as a sign bit. When the character is referenced it will be converted to a signed integer by extending that bit throughout the eight high-order bits that are appended to it. Therefore, we should not expect a character variable that is not declared unsigned to compare equally to the same character constant if the high-order bit is set. For more on this see Chapter 4.

String Constants

Strictly speaking, C does not recognize character strings, but it does recognize arrays of characters and provides a way to write constant character arrays which are called *strings*. Surrounding a character sequence with quotation marks ("") sets up an array of characters and generates the *address* of the array. In other words, at the point in a program where it appears, a string constant produces the *address* of the specified array of character constants. The array itself is located elsewhere. This is very important to remember. Notice that this differs from a character constant, which generates the value of the constant directly.

Just to be sure that this distinct feature of the C language is not overlooked, consider the following program:

```
main() {
        char *cp;
        cp = "hello world\n";
        printf(cp);
        }
```

The function **printf()** must receive the *address* of a string as its first (in this case, only) argument. The address of the string is assigned to the character pointer **cp.** Then the value of **cp** is passed to the function. Unlike other languages, the string itself is not assigned to **cp,** only its address. Because **cp** is a 16-bit object, it therefore cannot hold the string itself. The same program could be written as:

```
main() {
        printf("hello world\n");
        }
```

In this case, it is tempting to think that the string itself is being passed to **printf**(); but, as before, only the address is being passed.

Since strings may contain as few as one or two characters, they provide an alternative way of writing character constants in situations where the address, rather than the character itself, is needed.

It is a convention in C to identify the end of a character string with a null (zero) character. Therefore, C compilers automatically suffix character strings with such a terminator. Thus, the string **"abc"** sets up an array of four characters (**'a', 'b', 'c',** and zero) and generates the address of the first character for use by the program.

Full C compilers permit long strings to be split between lines, but Small C makes no such provision. See Chapter 28 for suggestions on how to add this capability to Small C; it is really quite easy.

Escape Sequences

Sometimes it is desirable to code nongraphic characters in a character or string constant. This can be done by using an *escape sequence*—a sequence of two or more characters in which the first (escape) character changes the meaning of the following character(s). When this is done the entire sequence generates only one character. C uses the backslash (\) for the escape character. The following escape sequences are recognized by the Small C compiler:

`\n`	newline
`\t`	tab
`\b`	backspace
`\f`	form feed
`\ooo`	value represented by the octal digits ooo

The term *newline* refers to a single character which, when written to an output device, starts a new line. Directed to a CRT screen it would place the cursor at the first column of the next line. When written to an output device or a character stream file, the newline character becomes a sequence of two characters: car-

riage return and line feed (not necessarily in that order). Conversely, on input a carriage return or a carriage return/line feed pair becomes a single newline character. Some implementations of C use the ASCII carriage return (13) as the newline character, while others use the ASCII line feed (10). It really doesn't matter which is the case, as long as we write **\n** in our programs. Avoid using the ASCII value directly, since that could produce compatibility problems between different compilers.

The sequence **\ooo** may be used to represent any character. It consists of the escape character followed by one, two, or three octal digits. The number ends when three digits have been processed, or a non-octal character is found.

There is one other type of escape sequence: anything undefined. If the backslash is followed by any character other than those described above, then the backslash is ignored and the following character is taken literally. So the way to code the backslash is by writing a pair of backslashes, and the way to code an apostrophe or a quote is by writing \' or \" respectively.

CHAPTER 4

Variables

The most significant limitation of the Small C compiler is its lack of support for many of the data types of the C language. It only recognizes integer and character variables. Although this may seem like a major limitation, Small C is, nevertheless, adequate for writing device drivers, compilers, assemblers, text processors, sort programs, and the like. Witness the Small C compiler itself.

A *variable* is a named object that resides in memory and is capable of being examined and modified. The term applies to pointers as well as integers and characters. Although arrays fit this definition, for reasons that appear in Chapter 6, they are better regarded as collections of variables. Pointers are treated in detail in Chapter 5.

Storage Classes

The term *storage class* refers to the method by which an object is assigned space in memory. Small C recognizes three storage classes: *static, automatic,* and *external.*

Statics Static variables are given space in memory at some fixed location within the program. They exist when the program starts to execute and continue to exist throughout the program's operation. The value of a static variable is maintained until we change it deliberately. At the assembly language level, each static variable has a label and is described to the assembler with a **DW** (define word) or **DB** (define byte) directive. The label consists of the variable's name (up to eight characters) prefixed by a compiler-generated underscore character. If a static integer, **si,** is to be loaded into the *accumulator register* (AX), it would be done with the assembly language instruction:

```
MOV AX,_SI
```

which instructs the CPU to copy the word at address **_SI** into AX. (See Appendix B for an overview of the 80x86 processors.)

The fact that they exist in permanently reserved memory locations means that static variables can be initialized to arbitrary values, which they are guaranteed to have at the beginning of program execution. (See Chapter 7 for more about initializing static objects.)

In Small C, global variables are always static, and only global variables are static. Full C, however, supports static locals.

Automatics Automatic variables, on the other hand, do not have fixed memory locations. They are dynamically allocated when the block in which they are defined is entered, and they are discarded upon leaving that block. Specifically, they are allocated on the machine stack by subtracting a value (one for characters and two for integers) from the stack pointer register (SP). Since automatic objects exist only within blocks, they can only be declared locally.

Because locals are allocated dynamically at unspecified memory (stack) locations, the program finds them by using the *base pointer* register (BP) to designate a *stack frame* (see Figure 8-1) for the currently active function. When the function is entered, the prior value of BP is pushed onto the stack, then the new value of SP is moved to BP. This address—the new value of SP—then becomes the base for references to local variables that are declared within the function.

If, for example, an integer **i** and a character **c** are defined (in that order) as automatic variables, the stack pointer would be decreased by three—two for **i** and one for **c.** If **i** is to be loaded into the accumulator, it would be done with the assembly instruction:

```
MOV AX,-2[BP]
```

which tells the CPU to locate the word addressed by BP minus 2, and copy its value into AX. Similarly, **c** would be obtained by:

```
MOV AL,-3[BP]
```

which locates the byte addressed by BP minus 3, and copies it into the lower half of AX.

Notice that this method of accessing local variables makes no use of labels, and that with each call of a function, its local variables may have different addresses depending on the stack address on entry to the function.

Obviously, when a local variable is created it has no dependable initial value. It must be set to an initial value by means of an assignment operation. Full C provides for automatic variables to be initialized in their declarations, like globals. It does this by generating "hidden" code that assigns values automatically after variables are allocated space. Small C does not support this feature; it requires the writing of assignment statements. This may seem like a shortcoming, but it really is not. For one thing, it takes very little more to write an assignment statement than an initializer (see Chapter 7). And, the result in the object program is the same; Small C is no less efficient for its lack of local initializers.

One last detail about automatics. When the current function is finished, it is necessary to restore BP and SP to their original values so the calling function will operate properly again. This is done by first moving BP to SP—to deallocate local variables—and then popping the original value of BP back into BP.

It is tempting to forget that automatic variables go away when the block in which they are defined exits. This sometimes leads new C programmers to fall into the "dangling reference" trap, in which a function returns a pointer to a local variable, as illustrated by:

```
func() {
int autoint;
...
return (&autoint);
}
```

When callers use the returned address of **autoint,** they will find themselves messing around with the stack space which **autoint** used to occupy.

In Small C, local variables are always automatic, and only local variables are automatic. Full C, however, supports static locals.

Externals Objects which are defined outside of the present source module have the *external* storage class. This means that, although the compiler knows *what* they are, it has no idea *where* they are. It simply refers to them by name without reserving space for them. Then when the linker brings together the object modules, it resolves these "pending" references by finding the external objects and inserting their addresses into the instructions that refer to them. The compiler knows an external variable by the keyword **extern** which must precede its declaration.

In Small C, only global declarations can be designated **extern,** and only globals in other modules can be referenced as external.

Scope

The *scope* of a variable is the portion of the program from which it can be referenced. We might say that a variable's scope is the part of the program that "knows" or "sees" the variable. As we shall see, different rules determine the scopes of global and local objects.

When a variable is declared globally (outside of a function) its scope is the part of the source file that follows the declaration—any function following the declaration can refer to it. Functions that precede the declaration cannot refer to it. Most C compilers would issue an error message in that case. However, Small C assumes that any undeclared name refers to a function, and automatically declares it as such.

The scope of local variables is the block in which they are declared. Local declarations must be grouped together before the first executable statement in the block—at the *head* of the block. It follows that the scope of a local variable effectively includes all of the block in which it is declared. Since blocks can be nested, it also follows that local variables are seen in all blocks that are contained in the one that declares the variables.

If we declare a local variable with the same name as a global object or another local in a superior block, the new variable temporarily supersedes the higher-level declarations. Consider the program in Listing 4-1.

```
char x;
main( ) {
        x = '1'; {
                char x;
                x = '2'; {
                        char x;
                        x = '3';
                        putc(x);
                        }
                putc(x);
                }
        putc(x);
        }
```

Listing 4-1. The Scope of Local Variables

This program declares variables with the name **x**, assigns values to them, and displays them on the screen in such a way that, when we consider its output, the scope of its declarations becomes clear. When this program runs, it displays **321**. This only makes sense if the **x** declared in the innermost block "masks" the higher-level declarations, so that **x** receives the value **'3'** without destroying the higher-level variables. Likewise, the second x is assigned **'2'** which it retains throughout the execution of the innermost block. Finally, the global **x**, which is assigned **'1'** is not affected by the execution of the two inner blocks. Notice, too, that the placement of the last two **putc(x);** statements demonstrates that leaving a block effectively unmasks objects that were hidden by declarations in the block. The second **putc(x);** sees the middle **x,** and the last **putc(x);** sees the global **x**.

This masking of higher level declarations is an advantage, since it allows the programmer to declare local variables for temporary use without regard for other uses of the same names.

Declarations

Unlike BASIC and FORTRAN, which will automatically declare variables when they are first used, every variable in C must be *declared* first. This may seem unnecessary, but when we consider how much time is spent debugging

BASIC and FORTRAN programs simply because misspelled variable names are not caught for us, it becomes obvious that the time spent declaring variables beforehand is time well spent.

As we saw in Chapter 1, describing a variable involves two actions—*declaring* its type, and *defining* it in memory (reserving a place for it). Although both of these may be involved, we refer to the C construct that accomplishes them as a *declaration.* As we saw above, if the declaration is preceded by **extern,** it only declares the type of the variables, without reserving space for them. In such cases, the definition must exist in another source file. Failure to do so will result in an "unresolved reference" error at link time.

Table 4-1 contains examples of legitimate variable declarations. Notice that the first two declarations are introduced by a keyword that states the data type of the variables listed. The keyword **char** declares characters, and **int** declares integers. This is the standard way to write declarations. Since it is not specified otherwise, both the character and integer variables declared by these statements are assumed by the compiler to contain signed values. We shall see in a moment that this assumption can be changed. The trend today is to make this assumption specific by placing the keyword **signed** before the data type; but Small C does not recognize that keyword.

The next three declarations begin with the prefix **unsigned,** which further qualifies the declaration by specifying unsigned treatment of the variables. When a declaration is introduced by such a qualifier, the data type may be omitted, causing the compiler to assume **int.**

As stated earlier, the ability to specify unsigned characters is new with this version of Small C. The original C language, and earlier versions of Small C, did not permit it. However, the trend in C compilers is to support this highly desirable feature, as the present version of Small C does.

Notice that Table 4-1 gives three examples of external declarations. Here too, when the data type is not given, the compiler assumes **int.**

When more than one variable is being declared, they are written as a list with the individual names separated by commas. Each declaration is terminated with a semicolon, as are all simple C statements.

Declaration	Comment
`int i;`	defines **i** and declares it to be an integer
`char x, y;`	defines **x** and **y** and declares them to be characters
`unsigned int ui;`	defines **ui** and declares it to be an unsigned integer
`unsigned u, v;`	defines **u** and **v** and declares them unsigned integers
`unsigned char uc;`	defines **uc** and declares it an unsigned character
`extern char z;`	declares **z** to be a character that is defined externally
`extern i, k;`	declares **i** and **k** to be integers, defined externally
`extern unsigned eu;`	declares **eu** to be an unsigned integer, defined externally

Table 4-1. Variable Declarations

As we shall see, this same basic syntax—with slight modifications—is used to declare pointers, arrays, and functions (see chapters 5, 6, and 8).

Integer Variables

Integers are 16-bit quantities. *Signed* integers are represented internally in two's complement notation. The high-order bit is a sign bit, and the 15 low-order bits are magnitude bits. This gives signed integers a positive range of 0 through 32767, and a negative range of -1 through -32768. As we saw above, a variable is understood to be signed if it is not explicitly declared unsigned.

Unsigned integers differ in that the high-order bit is taken as a magnitude bit. This gives unsigned integers a range of 0 through 65535.

When a signed integer enters into an operation with an unsigned quantity, the signed integer is treated as though it were unsigned. There is no actual change to its bit pattern, it is simply taken as an unsigned value. The result of such operations is also an unsigned value.

When a signed integer combines with another signed quantity, a signed operation is performed and the result is considered to be signed. In many cases there is no difference between signed and unsigned operations. Examples are addition and subtraction, and equality and inequality comparisons. Nevertheless, if either operand is unsigned, the result is unsigned.

Character Variables

Character variables are stored as 8-bit quantities. When they are fetched from memory, they are always promoted automatically to integers. This is the only automatic data conversion performed by Small C on fetched variables since integers are the largest values it recognizes. Full C likewise promotes characters to integers, but it may also perform further conversions, if needed, to match the other operand of a binary operation.

Originally, the C language did not distinguish between signed and unsigned character variables. It simply treated them as signed quantities. When it converted them to integers, it did so by extending the high-order (sign) bit throughout the high-order byte. But this approach often produces undesirable results when working with characters that have the high-order bit turned on. If, for example, a variable **ch** contains the value 0x80, one would expect the condition:

```
(ch == 0x80)
```

to yield *true*. But since the high-order bit of **ch** is set, the condition effectively becomes:

```
(0xFF80 == 0x80)
```

which is *false*. This forces the programmer to reset the eight high-order bits to zero by writing something like:

```
(ch & 0xFF == 0x80)
```

where **&** is the bitwise AND operator. Written in this form, the condition yields *true*.

Some compiler implementors have decided that it is better to treat character variables as unsigned values (like character constants). This eliminates the need to strip high-order bits and saves untold hours of debugging time. Today, most implementors still follow the UNIX convention, but also provide a compile-time option to treat unqualified character variables as unsigned. Most recently, the ability to designate character variables as either signed or unsigned has become standard.

Small C follows the UNIX convention by assuming that unqualified character variables are signed. However, the current version of Small C does accept the unsigned qualifier in character declarations to reverse this assumption.

Signed characters are represented internally in two's complement notation—the high-order bit being the sign, and the 7 low-order bits specifying the magnitude. This gives signed characters a positive range of 0 through 127, and a negative range of -1 through -128.

Unsigned characters differ in that the high-order bit is taken as a magnitude bit and the sign is always presumed to be positive. This gives unsigned characters a range of 0 through 255.

As with integers, when a signed character enters into an operation with an unsigned quantity, the character is interpreted as though it were unsigned. The result of such operations is also unsigned. When a signed character joins with another signed quantity, the result is also signed.

There is also a need to change the size of characters when they are stored, since they are represented in the CPU as 16-bit values. In this case, however, it does not matter whether they are signed or unsigned. Obviously there is only one reasonable way to put a 16-bit quantity into an 8-bit location—the high-order byte must be chopped off. It is the programmer's responsibility to ensure that significant bits are not lost when characters are stored.

CHAPTER 5

Pointers

The ability to work with memory addresses is an important feature of the C language. In many situations, array elements can be reached more efficiently through pointers than by subscripting. It also allows pointers and pointer chains to be used in data structures. This added degree of flexibility is always nice, but in systems programming it is absolutely essential.

Addresses and Pointers

Addresses that can be stored and changed are called *pointers*. A pointer is really just a variable that contains an address. Although they can be used to reach objects in memory, their greatest advantage lies in their ability to enter into arithmetic (and other) operations, and to be changed.

Not every address is a pointer. For instance, we can write **&var** when we want the address of the variable **var.** The result will be an address which is not a pointer since it does not have a name or a place in memory—it cannot, therefore, have its value altered.

Another example is an array name. As we shall see in Chapter 6, an unsubscripted array name yields the address of the array. But, since the array cannot be moved around in memory, its address is not variable. So, although such an address has a name, it does not exist as an object in memory (the array does, but its address does not) and cannot, therefore, be changed.

A third example is a character string. Chapter 3 indicated that a character string yields the address of the character array specified by the string. In this case the address has neither a name or a place in memory, so it too is not a pointer.

Pointer Declarations

The syntax for declaring pointers is like that for variables (see Chapter 4), except that pointers are distinguished by an asterisk that prefixes their names. Table 5-1 illustrates several legitimate pointer declarations. Notice in the third example that we may mix pointers and variables in a single declaration. Also notice that the data type of a pointer declaration specifies the type of object to which the pointer refers, not the type of the pointer itself. As we shall see, all Small C pointers (integer or character) contain 16-bit *segment offsets*.

The best way to think of the asterisk (*) is to imagine that it stands for the phrase *object at* or *object pointed to by*. The first declaration in Table 5-1 then reads *the object at (pointed to by)* **ip** is an integer.

Declaration	Comment
`int *ip;`	defines **ip** and declares it to be a pointer to integers
`char *cp;`	defines **cp** and declares it to be a pointer to characters
`unsigned u, *uptr;`	defines **u** and **uptr** and declares **u** to be an unsigned integer and **uptr** to be a pointer to unsigned integers
`unsigned char *ucp;`	defines **ucp** and declares it to be a pointer to unsigned characters
`extern unsigned *ep;`	declares **ep** to be a pointer (defined externally) to unsigned integers

Table 5-1. Pointer Declarations

8086 Memory Addressing

The size of a pointer depends on the architecture of the CPU and the implementation of the C compiler. The 8086 CPU incorporates a segmented memory-addressing scheme in which an effective address is composed of two parts. There is a 16-bit *segment address* and the 16-bit *offset* within the segment. The CPU adds these parts together, with the segment address shifted left four bits, to derive the effective address for a memory reference. (See Appendix B for details, especially figures B-2 through B-4.)

To be perfectly general, an 8086 address must occupy two 16-bit words—one for the segment address and one for the offset. But, since a given segment is usually referenced frequently, while transitions between segments are relatively rare, the CPU contains *segment registers* (Figure B-1) that "remember" the segment addresses. Therefore, it is not always necessary for an address to specify more than the 16-bit offset.

The flexibility of this addressing scheme allows programs to utilize various schemes for segmenting memory so as to obtain a suitable compromise between program size and efficiency. Notice that a 16-bit offset effectively restricts segments to 64K bytes or less—large programs must be divided into multiple segments. In 8086 terminology, addresses that specify both segment and offset are called *far* addresses, and those that provide only the offset are called near addresses.

Although some C compilers permit us to designate any of several memory-addressing schemes *(memory models),* Small C always uses just one. Figure 5-1 illustrates the way Small C programs use memory, and the segment and stack pointer registers.

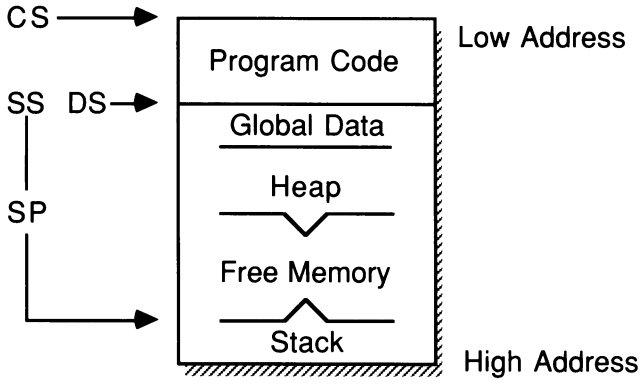


Figure 5-1. Small C Memory Model

Each Small C program is divided into two segments: a *code segment,* which is addressed by the CPU's *code segment register* (CS); and a *data segment,* which is addressed by the *data segment register* (DS). The stack segment and data segment are one and the same, so the stack segment register (SS) contains the same address as DS. These segment register addresses remain fixed throughout the execution of a program, so only near pointers are needed. Therefore, pointers are always just 16-bit offsets in Small C programs.

The code segment is only as large as it has to be to contain the program's instructions. The data/stack segment is exactly 64K bytes in size, if that much memory is available when the program begins execution. If less is available, then whatever is available is used. Notice how the data/stack segment is organized. Globally declared data occupies a fixed space at the low (address) end. Following that is the *heap*—space that has been allocated by calling the library functions **malloc()** and **calloc().** The heap expands and contracts as program execution progresses. At the high end is the stack, which grows toward the low end of the segment and shrinks back into the high end as the program executes. This leaves the area in the middle free for either heap or stack use. These last two areas should never overlap; if they do the program will certainly misbehave, and

may well go berserk. The library function **avail()** returns the amount of free memory available and optionally aborts the program if the stack overlaps allocated memory.

Pointers and Integers

As the preceding discussion indicates, Small C pointers occupy one word, just as integers do. This is fortunate, since Small C does not support arrays of pointers. In Small C, we are free to store pointer values in integer arrays; the compiler will not complain. This sort of "abuse" of data types used to be common in C circles, but these days it is frowned upon and the newer compilers gripe about it. Personally, I find it only slightly irksome, since it obviates a major Small C shortcoming. A consequence of this practice is that when it becomes necessary to use a pointer in an integer array, we must first assign it to a pointer variable so the compiler will know that it is working with an address rather than an integer.

Pointer Arithmetic

Another major difference between addresses and ordinary variables or constants has to do with the interpretation of addresses.

Since an address points to an object of some particular type, adding one (for instance) to an address should direct it to the next *object,* not necessarily the next *byte*. If the address points to integers, then it should end up pointing to the next integer. But, since integers occupy two bytes, adding one to an integer address must actually increase the address by two. A similar consideration applies to subtraction. In other words, values added to or subtracted from an address must be scaled according to the size of the objects being addressed. This automatic correction saves the programmer a lot of thought and makes programs less complex since the scaling need not be coded explicitly. Notice that while the scaling factor for integers is two, for characters it is one; therefore, character addresses do not receive special handling. It should be obvious that if objects of other sizes were supported, then different factors would have to be used.

A related consideration arises when we imagine the meaning of the differ-

ence of two addresses. Such a result is interpreted as the number of objects between the two addresses. If the objects are integers, the result must be divided by two in order to yield a value which is consistent with this meaning. See Chapter 6 for more on address arithmetic.

When an address is operated on, the result is always another address. Thus, if **ptr** is a pointer, then **ptr+1** is also an address.

The Indirection Operator

The asterisk (*), which we saw in pointer declarations, can also be used as an operator in expressions. We call it the *indirection operator* because when it precedes a pointer (or any address) it produces an indirect reference to the indicated object. Thus, if **ip** is a properly initialized integer pointer, then:

```
*ip
```

yields the integer pointed to by **ip.** On the other hand:

```
ip
```

yields the address contained in the pointer.

At the assembly-language level, an indirect reference is accomplished by first loading the value of the pointer into a register, and then using it to fetch the object. If **ip** is a global pointer to integers, then ***ip** generates:

```
MOV BX,_IP
MOV AX,[BX]
```

The first instruction loads the contents of **ip** into BX, and the second loads the word addressed by BX into AX. It should be clear why this is called an *indirect* reference.

Pointer Comparisons

One major difference between pointers and other variables is that pointers are always considered to be unsigned. This should be obvious, since memory addresses are not signed. This property of pointers (actually all addresses) ensures that only unsigned operations will be performed on them. It further means that the other operand in a binary operation will also be regarded as unsigned (whether or not it actually is). For instance, if (as we saw above) an integer array **ia[]** actually contains addresses, then it would make sense to write:

```
(ia[5] > ptr)
```

which performs an unsigned comparison, since **ptr** is a pointer. Thus, if **ia[5]** contains -1 and **ptr** contains 0x1234, the expression will yield *true,* since -1 is really 0xFFFF—a higher unsigned value than 0x1234. So, although the array is thought by the compiler to contain signed integers, the proper type of comparison is performed anyway. (Note: With the current version of Small C, we could designate the array **unsigned** for better documentation.)

It makes no sense to compare a pointer to anything but another address or zero. C guarantees that valid addresses can never be zero, so that particular value is useful in representing the absence of an address in a pointer.

Furthermore, to avoid portability problems, only addresses within a single array should be compared for relative value (as above). To do otherwise would necessarily involve assumptions about how the compiler organizes memory. Comparisons for equality, however, need not observe this restriction, since they make no assumption about the relative positions of objects.

An Example

A final example may help pull the ideas in this chapter together. Consider the program fragment in Listing 5-1:

```
char *cp, *cend;
int  *ip;
...
cend = cp + 5;
while (cp < cend) {
       *ip = *ip + *cp;
       ip = ip + 1;
       cp = cp + 1;
       }
```

Listing 5-1. Example of the Use of Pointers

This code adds the values of five characters to corresponding integers. First, the pointer **cend** is set five characters beyond the address in **cp** (which we presume has already been initialized properly). The **while** statement then repeatedly tests whether **cp** is less than **cend.** If so, the compound statement is performed. If not, control passes to whatever follows. With each execution of the compound statement, the object at **cp** (a character) is added to the object at **ip** (an integer) with the object at **ip** receiving the result. Then both **cp** and **ip** are incremented to the next objects. Since **ip** is an integer pointer, each increment advances it two bytes to the next integer. The loop executes five times. After the task is finished, **cp** is no longer less than **cend,** and control goes on to whatever follows.

CHAPTER 6

Arrays

An *array* is a collection of like variables that share a single name. The individual *elements* of an array are referenced by appending a subscript, in square brackets, behind the name. The subscript itself can be any legitimate C expression that yields an integer value—that includes every Small C expression. Most programming languages provide for arrays of more than one dimension; that is, each element may itself be a sub-array of smaller elements. Small C, however, only supports single-dimension arrays. Therefore, arrays in Small C may be regarded as simple linear collections of like variables.

Array Subscripts

When an array element is referenced, the subscript expression designates the desired element by its position in the array. The first element occupies position zero, the second position one, and so on. It follows that the last element is subscripted by [n-1] where n is the number of elements in the array. The statement

```
array[5] = 0;
```

for instance, sets the sixth element of **array** to zero.

Since array subscripts are treated as integers, they are, in fact, signed values. It is the programmer's responsibility to see that only positive values are produced, since a negative subscript would refer to some point in memory preceding the array. As we shall see, there is a place for negative subscripts, which are not associated with array names.

Array Declarations

The number of elements in an array is determined by its declaration. Appending a constant expression in square brackets to a name in a declaration identifies the name as the name of an array with the number of elements indicated. The examples in Table 6-1 are valid declarations.

Declaration	Comment
`char ca[8];`	Declares **ca** to be an array of characters and defines eight bytes of storage for it.
`int ia1[25], ia2[SZ + 1];`	Declares **ia1** to be an array of 25 integers and **ia2** to be an array of **(SZ + 1)** integers, where **SZ** is defined as a constant expression. Twenty-five plus **(SZ + 1)** words of space are defined.
`unsigned u, ua[20];`	Declares and defines **u** to be an unsigned integer, and **ua** to be an array of 20 unsigned integers.
`extern int ia[];`	Declares **ia** to be an integer array which is defined externally.

Table 6-1. Array Declarations

Notice in the third example that ordinary variables may be declared together with arrays in the same statement. In fact, array declarations obey the syntax

rules of ordinary declarations, as described in chapters 4 and 5, except that certain names are designated as arrays by the presence of a dimension expression.

What about the undimensioned array in Table 6-1? How can the compiler work with non-specific array sizes? This brings up an important fact about the C language. In C, array dimensions serve only to determine how much memory to reserve; it is the programmer's responsibility to stay in proper bounds when referencing array elements. So it is not necessary for the compiler to know the size of an array at the point where it is referenced. It only needs that information at the point of definition—where storage is actually reserved for it. Therefore, in declarations that do not define the array, the dimension expression is useless; all that is needed are the brackets to designate the name as an array name. There are two such situations—external declarations and declarations of arguments in function headers. We shall see in Chapter 8 that a function argument declared as **arg[]** is really a pointer.

Another situation in which an array's size need not be specified is when the array elements are given initial values. As we will see in Chapter 7, the compiler will determine the size of such an array from the number of initial values.

Array References

In C, we may refer to an array in several ways. Most obviously, we can write subscripted references to array elements, as we have already seen. Not so obvious is C's reaction to the appearance of an unadorned array name. C interprets an unsubscripted array name as the *address* of the array. We could, for instance, set a pointer to the address of an array with the expression:

```
pointer = array;
```

This interpretation of array names leads to even more interesting uses. In some ways, we can use array names like pointers. We can legally write:

```
i = *array;
```

This assigns the *object at* **array** to **i**. Since the address of an array is also the address of its first element, this statement copies the first element of **array** to **i.** More will be said about this later.

Chapter 5 introduced the address operator **&** which yields the address of an object. This operator may also be used with array elements. Thus, the expression **&array[3]** yields the address of the fourth element of **array.** Notice, too, that **&array[0], array+0,** and **array** are all equivalent. It should be clear by analogy that **&array[3]** and **array+3** are also equivalent.

To refer to an array element, the Small C compiler adds the subscript (scaled in the case of integer arrays) to the address of the array. The result points to the desired object. This sequence of steps suggests an alternate way for programmers to reference array elements—by adding subscripts to array names and applying the indirection operator to the result. Thus, **array[x]** is equivalent to ***(array+x).** The parentheses are required, since without them the expression would be evaluated as **(*array)+x,** which is quite different. Instead of fetching the element at **array+x,** it would add **x** to the first element of **array.**

It follows from this discussion that the first element of **array** may be written as **array[0], *(array+0),** or ***array.**

Pointers and Array Names

These considerations suggest that pointers and array names might be used interchangeably. And, in many cases, they may. C will let us subscript pointers and also use array names as addresses (as mentioned above). Assuming that the integer pointer **ip** contains the address of an array of integers, it is perfectly reasonable and legal to write either **ip[4]** or ***(ip+4).** On the other hand, as we saw above, **ip** could be the name of an array.

It is important to realize that although C accepts unsubscripted array names it does not accept them everywhere. We cannot write:

```
array = y;
```

What sense would that make? It asserts that the value in **y** is to be assigned as the address of **array.** But an array, like any object, has a fixed home in memo-

ry—its address cannot be changed. We say that **array** is not an *lvalue;* that is, it cannot be used on the *left* side of an assignment operator (nor may it be operated on by increment or decrement operators); it simply cannot be changed. Not only does this assignment make no sense, it is physically impossible, because an array address is not a variable. There is no place reserved in memory for an array's address to reside; there is only space for its elements.

On the other hand, an array name may appear in a larger expression that is an lvalue. Examples are:

```
*array = x;
```

and

```
array[0] = x;
```

where the subexpressions ***array** and **array[0]** are, in fact, lvalues.

Negative Subscripts

The previous section makes it plain that a pointer may refer to an array. For instance, we might write a function that performs some operation on any array. A pointer to the pertinent array would be passed to the function, and it would do its thing without caring which array it is working with. But, since a pointer may address any element of an array—not just the first one—it follows that negative subscripts applied to pointers might yield array references that are in bounds. This sort of thing might be useful in situations where there is a relationship between successive elements in an array, and it becomes necessary to reference an element preceding the one being pointed to.

Address Arithmetic

As we have seen, addresses (pointers, array names, and values produced by the address operator) may be used freely in expressions. This one fact is responsible for much of the power of C as a systems programming language.

As with pointers (see Chapter 5), all addresses are treated as unsigned quantities. Therefore, only unsigned operations are performed on them. Furthermore, the opposite operand in a binary operation is regarded as unsigned (whether or not it actually is). The results of such operations are also considered unsigned, as they participate in the further process of expression evaluation.

Of all the arithmetic operations that could be performed on addresses, only two make sense: displacing an address by a positive or negative amount, and taking the difference between two addresses. All others, though permissible, yield meaningless results.

Displacing an address can be done either by means of subscripts or by use of the plus and minus operators, as we saw earlier. These operations should be used only when the original address and the displaced address refer to positions in the same array or data structure of some sort (an operating system control block for instance). Any other situation would assume a knowledge of how memory is organized and would, therefore, be ill-advised for portability reasons.

As we saw in Chapter 5, taking the difference of two addresses is a special case in which the compiler interprets the result as the number of objects lying between the addresses. It should be clear that certain nonsense expressions of this type might be written. Examples are:

1. taking the difference of a character address and an integer address
2. taking the difference of addresses which are not in the same array
3. subtracting an address from a smaller address

The compiler will accept these, but the results will not be useful.

An Example

The example in Listing 5-1 may be rewritten using array notation. Listing 6-1 shows the result. First, integer **i** is set to zero. Then the **while** statement controls repeated execution of a compound statement which does the work. As long as **i** is less than 5, corresponding elements of the arrays **ca[]** and **ia[]** are added with the result going into **ia[].** With each iteration, **i** is incremented by 1. When **i** becomes 5, control leaves the **while** and goes on to whatever follows.

```
char ca[5];
int ia[5], i;
...
i = 0;
while (i < 5) {
ia[i] = ia[i] + ca[i];
i = i + 1;
}
```

Listing 6-1. Example of the Use of Arrays

CHAPTER 7

Initial Values

Most programming languages provide ways of specifying *initial values;* that is, the values that variables have when program execution begins.

By designating initial values, we avoid having to write assignment statements to initialize variables, and eliminate from the executable program the code which they would generate. The main advantage of eliminating this code is a reduction in program size. There is also a speed advantage; however, since initializing assignments are executed only once, the time saved is usually insignificant.

Initial Values and Serial Reusability

Some operating systems provide an environment in which a program may be invoked any number of times after being loaded into memory once. The memory-resident utilities that run on MS-DOS machines are an example. In such cases, the programmer must ensure that subsequent invocations of the program will see the same initial values as the first invocation. When that is done, the program is said to be *serially reusable.* In other words, to be serially reusable, a program's behavior must not be influenced by previous executions.

If not obvious, it should at least seem reasonable that relying on initializers, instead of assignment statements, will not provide serial reusability. The reason is that when the compiler processes a declaration with an initial value, it tells the assembler to preset the memory for the variable with the specified value. After the first execution of the program, the variable will be left with its final value. Then, when the program is invoked again, it finds the final value instead of the required initial value. And that leads to faulty performance.

Therefore, if serial reusability is important, we should use assignment state-

ments to set preliminary values for any variables which might be changed by program execution. Then, with each subsequent execution, the variables will be reassigned their preliminary values.

Globals versus Locals

Small C implements some, but not all, of the initializing capabilities of the Full C language. Small C initializes only globals, whereas Full C initializes both globals and locals.

The ability to initialize locals produces only an illusion of efficiency. This is because automatic objects are created at the time control reaches their declarations. If they are given initial values, then there must be hidden assignment statements to give them their designated values. So, while it may look efficient at the source level, there is no real advantage at the object level. Neither is there much advantage at the source level, since writing assignment statements takes very little more effort than writing initializers, as we shall see. Arrays are the exception, since either an iterative statement or a list of statements must be written.

Uninitialized Small C globals (static storage) start life at zero, and all Small C locals (automatic storage) initially have unpredictable values.

Initializer Syntax

Specifying initial values is simple. In its declaration, we follow a variable's name with an equal sign and a constant expression for the desired value. Character constants with backslash escape sequences are permitted. Thus,

```
int i = 80;
```

declares **i** to be an integer, and gives it an initial value of 80. And,

```
char ch = '\n';
```

declares **ch** to be a character, and gives it the value of the newline character.

If array elements are being initialized, a list of constant expressions, separated by commas and enclosed in braces, is written. For example,

```
int ia[3] = {1, 2, 3};
```

declares **ia** to be an integer array, and gives its elements the values 1, 2, and 3, respectively. If the size of the array is not specified, it is determined by the number of initializers. Thus,

```
char ca[] = {'a', 0};
```

declares **ca** to be a character array of two elements which are initialized to the lowercase letter **a** and zero, respectively. On the other hand, if the size of the array is given, and if it exceeds the number of initializers, the leading elements are initialized and the trailing elements default to zero. Therefore,

```
int ia[3] = 1;
```

declares **ia** to be an integer array of three elements, with the first element initialized to 1 and the others to zero. Finally, if the size of an array is given and there are too many initializers, the compiler generates an error message. In that case, the programmer must be confused.

Character arrays and character pointers may be initialized with a character string. In these cases, a terminating zero is automatically generated. For example,

```
char ca[4] = "abc";
```

declares **ca** to be a character array of four elements with the first three initialized to **a, b,** and **c,** respectively. The fourth element contains zero. If the size of the array is not given, it will be set to the size of the string plus one. Thus, **ca** in:

```
char ca[] = "abc";
```

also contains the same four elements. If the size is given and the string is shorter, trailing elements default to zero. For example, the array declared by:

```
char ca[6] = "abc";
```

contains zeroes in its last three elements. If the string is longer than the specified size of the array, the array size is increased to match. If we write:

```
char *cp = "abc";
```

the effect is quite different from initializing an array. First, a word (16 bits) is set aside for the pointer itself. This pointer is then given the address of the following byte. Then, beginning with that byte, the string and its zero terminator are assembled. The result is that **cp** contains the address of the **"abc"** string.

Small C accepts initializers for character variables, pointers, and arrays, and for integer variables and arrays. The initializers themselves may be either constant expressions, lists of constant expressions, or strings. But not all combinations of object type and initializer type are legal. Table 7-1 shows which combinations are supported. Small C supports an upward-compatible subset of the Full C initializers.

		constant expression	list of constant expressions	character string
character	variable	yes		
	pointer		yes	yes
	array	yes	yes	yes
integer	variable	yes		
	pointer			
	array	yes	yes	

Table 7-1. Permitted Object/Initializer Combinations

CHAPTER 8

Functions

The notion of *functions* in programming languages is based on the mathematical concept of the same name. The idea is that a named algorithm—a function—maps a set of values onto *one* of a set of result values. When the algorithm's name is written in an expression, together with the values it needs, it represents the result that it produces. In other words, an operand in an expression may be written as a function name, together with a set of values upon which the function operates; the resulting value, as determined by the function, replaces the function reference in the expression. For example, in the expression:

```
abc(a, 2) + 1
```

the term **abc(a, 2)** names the function **abc,** and supplies the variable **a** and the constant **2** from which **abc** derives a value, which is then added to **1** for the final value of the expression. If the algorithm of **abc** happens to form the product of its two values, then the expression effectively becomes:

```
(a * 2) + 1
```

Despite the simplicity of this illustration, functions are very powerful devices. Their power comes from two sources: although a function may be invoked any number of places in a program, it is defined only once; also, functions may have arbitrarily complex algorithms. It follows from the first fact that functions save work (programming effort) and memory space. And both facts together imply that functions are an effective structuring device for organizing programs into logical units. The trade-off, of course, is the overhead of performing the requisite function calls and returns.

Functions and Subroutines

Programming languages that support functions implement them as special subroutines which return a value and are called from within expressions. They refer to them as functions to distinguish them from ordinary subroutines (or *procedures*) that do not return values and are not called from within expressions. The C language uses functions as the basic procedural units of programs. In fact, C does not even implement ordinary subroutines. This may seem to be a major shortcoming. But, as we shall see, it is really an elegant simplification that loses none of the generality of other languages.

C is unique in that it accepts an isolated expression as a statement. Expressions need not be written as parts of larger constructs. Thus we could write **i+5;** or just **5;** as independent statements if we wanted to. What is the point? What do such statements accomplish? These particular statements accomplish nothing. But when we consider that expressions may invoke functions, and may also include assignment, increment, and decrement operations, then it becomes clear that expressions are capable of doing work beyond that of simply yielding a value.

C itself does not even recognize *assignment* statements as such. Assignment statements are really just expressions containing assignment operators. Furthermore, while the concept of function is essentially that of an algorithm yielding a value, functions are not limited to this narrow definition. There is no reason that functions cannot do other things which affect program execution (interact with the operating system, for instance).

These considerations lead to the observation that C has no need for a *call* statement. To "call" a function, it is only necessary to write an expression consisting of nothing more than a function reference; that is, the function's name followed by parentheses containing the values to be passed to it. Thus, the expression:

```
func();
```

is a perfectly valid statement that calls the function **func().** Although no val-

ues are passed to **func(),** it is still necessary to write the parentheses so the compiler will know to interpret this as a call to **func().** Without the parentheses, this would be taken as a reference to the function's address.

By eliminating the distinction between functions and subroutines, it would seem that C creates ambiguities relating to the handling of *returned values.* Do all functions return a value? And, if so, what happens to the value of a function like **func(),** which is called without a context to utilize its value? The answer to the first question is no, not all functions actually return a value. On the other hand, **func()** may, in fact, return a value which in this case is wasted—since it is not used it is simply ignored. The only dangerous situation occurs when a function that does not return a value is called from a context that requires one. Suppose, for instance, that the function **novalue(),** which does not return a value, is called in the statement:

```
if (novalue()) i = 5;
```

Will the assignment be performed? The answer is "maybe so, maybe not." When a function is written to not return a value, the last value that happens to be in Small C's primary register (the AX register) will be seen as the returned value when it returns to the point from which it was called. So, functions that do not return a value effectively return unpredictable values which should be ignored when the function is called.

Terminology

Over the years, accepted terminology concerning the implementation of functions in programming languages has accrued. To begin with, we refer to the part of a program that describes a function as a function *declaration* or *definition.* And, as we saw in Chapter 1, C distinguishes between these terms. It will become clear momentarily that we may declare a function's existence without actually defining its algorithm.

The values on which a function works are called *arguments.* Arguments appear in two contexts—in function *definitions* and in function *calls.* In the first case, the arguments are just names by which reference is made to the values that

are actually passed to the function when it is called. We call these *formal* or *dummy* arguments. In the second case, the arguments specify what is actually passed to the function, and so they are called *actual* arguments. We have already been using three terms that need to be identified—*call* refers to the act of invoking a function; *return* refers to the return of control from a function back to the point from which it was called; and *returned value* refers to the value produced by a function.

Functions may affect programs in ways other than through their returned values. We call these actions *side effects*. Typically, side effects take the form of changes to global objects, changes to objects pointed to by arguments, and input/output operations.

Function Declarations

As we noted before, the distinction C makes between declarations and definitions applies to functions as well. Declarations, which specify the contents of a function, are said to *define* the function. On the other hand, there are situations where it is necessary to only *declare* a function's existence and the type of value it returns. We now consider declarations of the latter type. Definitions will be discussed next.

Earlier, we noted that it was the appearance of parentheses following a function's name that informed the compiler to generate a call to the function. That same method is used to distinguish function declarations from other types of declarations. Thus, the first declaration in Table 8-1 identifies **abc** as the name of a function.

Declaration	Comment
`int abc();`	Declares **abc** to be a function which returns an integer.
`int (*fp)();`	Declares **fp** to be a pointer to a function which returns an integer.
`extern func();`	Declares **func** to be an externally defined function which returns an integer.

Table 8-1. Function Declarations

The second declaration in Table 8-1 looks a bit complicated because it has two sets of parentheses and an asterisk. In fact, it only declares **fp** to be a pointer to any function that returns integers. As in other declarations, the asterisk identifies the following name as a pointer. Therefore, this declaration reads *fp is a pointer to a function returning int.* Using the term *object* loosely, the asterisk may be read in its usual way as *object at.* Thus, we could also read this declaration as *the object at fp is a function returning int.*

So why the first set of parentheses? By now you have noticed that in C, declarations follow the same syntax as references to the declared objects. And, since the asterisk and parentheses (after the name) are expression operators, an evaluation precedence is associated with them. In C, parentheses following a name are associated with the name before the preceding asterisk is applied to the result. Therefore,

```
int *fp();
```

would be taken as:

```
int *(fp());
```

which says that *fp is a function returning a pointer to an integer,* which is not

at all like the second declaration in Table 8-1.

I do not mean to imply that Small C will accept these last two examples. Among Small C's restrictions is its limit of only one *declaration modifier* (asterisk, parentheses, etc.) per name. So, although these two declarations are acceptable C syntax, Small C will reject them. The only exception is pointers to functions as shown in Table 8-1.

The last example in Table 8-1 shows how to declare functions that are defined in a different source file that is compiled separately.

Whereas most C compilers allow us to also declare (in non-definitional declarations) the types of the formal arguments, Small C does not. The parentheses must be empty in Small C programs. The reason some compilers permit this is so they can ensure that arguments are passed to functions correctly. These prototype declarations are a very useful debugging feature. Chapter 28 discusses the possibility of adding prototype declarations to the Small C compiler.

There are three contexts in which non-definitional function declarations may appear—global declarations, local declarations, and declarations of formal arguments (below). Small C does not accept every form of declaration in Table 8-1 in every context. The first and third types of declaration may be written only at the global level, whereas the second type (pointer to function) may appear either in local or formal argument declarations.

Of the three types of function declaration in Table 8-1, only one is ever required in a Small C program—the pointer declaration. Unlike most C compilers, when Small C encounters an undefined name, it presumes a function name. And, since Small C functions return only integers, the compiler is able to automatically declare such names as functions returning integers. This means that we may freely reference functions in a source file before defining them. At the end of the source file, if any of these automatically declared functions have not been defined, then code is generated which declares them to the assembler as external references. It follows, therefore, that the first and third examples of Table 8-1 are never needed in Small C programs. They are supported only for documentational purposes and for compatibility with Full C compilers.

Although Small C does not support functions that return characters, it does

accept such declarations. This is because, since characters are automatically promoted to integers, there is no practical difference between functions returning integers and those returning characters. So, when Small C sees **char func();** it accepts it as **int func();.**

Function Definitions

The second way to declare a function is to fully describe it—that is, to define it. Obviously every function must be defined somewhere. Small C function definitions have the form

```
Name (ArgumentList?)
  ArgumentDeclaration?...
  CompoundStatement
```

Name is the name of the function. ***ArgumentList?*** is a list of zero or more names for the arguments that will be received by the function when it is called. These are the *formal (dummy)* arguments mentioned earlier. They simply tag the *actual* arguments with names by which they can be referenced within the function. The first actual argument corresponds to the first formal argument, the second with the second, and so forth.

ArgumentDeclaration?... is a series of declarations which specify the attributes of the formal arguments. Each name in ***ArgumentList?*** must be declared. These are ordinary declarations of variables, pointers, arrays, and function pointers as described in Chapters 4 through 6 and previously in this chapter. In this context, however, the declared items are not defined; instead, they are presumed to exist on the stack—below the return address—where they are placed before the function is called.

Since every actual argument is passed as a 16-bit value, characters are promoted to integers (as usual), and arrays and strings are passed as pointers. This is entirely consistent with their treatment in expressions. In fact, actual arguments are just that—expressions.

Although a character is passed as a word, we are free to declare its formal argument as either character or word. If it is declared as a character, only the

low-order byte of the actual argument will be referenced, and (as usual) will be promoted to an integer. If it is declared as an integer, then all 16 bits will be referenced. The only difference is the point at which the promotion to integer occurs. In the first case, it occurs both at the point of the call and at the points of reference in the function being defined (according to the argument's formal declaration—unsigned or signed). In the second case, it occurs only at the point of the call, where the actual argument is pushed onto the stack. In that case the promotion is regulated by the actual argument's declaration. Of course, the argument may be a character constant, in which case the promotion occurs at compile time, such that the value is positive.

It is generally more efficient to reference integers than characters because there is no need for a machine instruction to set the high-order byte. So it is common to see situations in which a character is passed to a function which declares the argument to be an integer. But there is one caveat here: not all C compilers promote character arguments to integers when passing them to functions; the result is an unpredictable value in the high-order byte of the argument. This should be remembered as a portability issue.

Since there is no way in C to declare strings, we cannot declare formal arguments as strings, but we can declare them as character pointers or arrays. In fact, as we have seen, C does not recognize strings, but arrays of characters. The string notation is merely a shorthand way of writing a constant array of characters.

Furthermore, since an unsubscripted array name yields the array's address, and since arguments are passed by value (below), an array argument is effectively a pointer to the array. It follows that the formal argument declarations **arg[]** and ***arg** are really equivalent. The compiler takes both as pointer declarations. Array dimensions in argument declarations are ignored by the compiler since the function has no control over the size of arrays whose addresses are passed to it. It must either assume an array's size, receive its size as another argument, or obtain it elsewhere.

The last, and most important, part of the function definition above is ***CompoundStatement.*** This is where the action occurs. Since compound state-

ments may contain local declarations, simple statements, and other compound statements, it follows that functions may implement algorithms of any complexity and may be written in a structured style. Nesting of compound statements is permitted without limit.

As an example of a function definition, consider:

```
func (i, ia, c, cp, fp)
       int i, ia[], (*fp)();
       char c, *cp;
       {...}
```

Here is a function named **func** which takes five arguments: an integer, an integer array, a character, a character pointer, and a function pointer. The ellipsis stands for whatever constitutes the function's algorithm. Notice that each argument is declared and only arguments are declared between the argument list and the compound statement. The order of the declarations is immaterial. This function without arguments would be declared as:

```
func () {...}
```

Notice that these definitions do not specify whether or not a function returns a value or the type of value returned. Full C compilers accept keywords like **int, char,** and **void** before a function definition to provide this information. The word **void** is a recent innovation which explicitly states that the function does not return a value. This version of Small C accepts void as a comment, but does not accept int or char.

For every function definition, Small C generates an assembler directive declaring the function's name to be *public*. This means that every Small C function is a potential *entry point,* and therefore can be accessed externally.

Function Calls

A function is called by writing its name, followed by a parenthesized list of argument expressions. The general form is:

```
Name (ExpressionList?)
```

where ***Name*** is the name of the function to be called. As indicated by the question mark, ***ExpressionList?*** is optional. Notice that each argument is an expression. It may be as simple as a variable name or a constant, or it may be arbitrarily complex, including perhaps other function calls. Whatever the case, the resulting value is pushed onto the stack where it is passed to the called function.

Small C programs evaluate arguments from left to right, pushing them onto the stack in that order. Then they call the function. On return, the arguments are removed from the stack and execution continues with the primary register containing the value returned by the function.

When the called function receives control, it refers to the first actual argument using the name of the first formal argument. The second formal argument refers to the second actual argument, and so on. In other words, actual and formal arguments are matched by position in their respective lists. Extreme care must be taken to ensure that these lists have the same number and type of arguments. Small C does not verify that functions are called properly.

It was mentioned earlier that function calls appear in expressions. But, since expressions are legal statements, and since expressions may consist of only a function call, it follows that a function call may be written as a complete statement. Thus, the statement:

```
func (x, y+1, 33);
```

is legal. It calls **func(),** passing it three arguments—**x, y+1,** and **33.** Since this call is not part of a larger expression, any value that **func()** might return will be ignored.

As another example, consider:

```
x = func ();
```

which is also an expression. It calls **func()** without arguments and assigns the returned value to **x.** If **func()** should fail to explicitly return a value, then an unpredictable value will be assigned.

The ability to pass one function of a pointer to another function is a very powerful feature of the C language. It enables a function to call any of several other functions, with the caller determining which subordinate function is to be called. We shall see this used to good effect in the Small C expression analyzer.

As an example of this technique, consider three functions—**func(), f1(),** and **f2()**—where **func()** is to call either **f1()** or **f2()** depending on instructions from its caller. If we call **func()** with:

```
func (f1);
```

it will know to call **f1()** instead of **f2().** (Recall that a function name, like **f1,** without parentheses, yields the function's address.) Now, if **func()** is defined as

```
func (arg) int (*arg)(); {
        ...
        (*arg)();
        ...
        }
```

it will perform properly. Notice that **arg** is declared to be a function pointer. Also, notice that the designated function is called by writing an expression of the same form as the declaration. Although not strictly necessary to make sense of this expression, the asterisk loosely meaning object at appearing before arg refers to the function itself. The first set of parentheses must be written as shown so the compiler will not apply the asterisk to the value returned by the call, instead of the function pointer.

This syntax, which was introduced with version 2.1 of Small C, is consistent with full C. Originally, Small C accepted:

```
int arg;
```

to declare such an argument, and:

```
arg (...)
```

to call the function. Although Small C still accepts this notation, to avoid portability problems it should not be used.

Argument Passing

Now let us take a closer look at the matter of argument passing. With respect to the method by which arguments are passed, two types of subroutine calls are used in programming languages—*call by reference* and *call by value*.

The first method passes arguments in such a way that references to the formal arguments become, in effect, references to the actual arguments. In other words, *references* (pointers) to the actual arguments are passed, instead of copies of the actual arguments themselves. In this scheme, assignment statements have implied *side effects* on the actual arguments; that is, variables passed to a function are affected by changes to the formal arguments. Sometimes side effects are beneficial, but usually they are not. Frequently, with this approach, it becomes necessary to declare temporary variables into which the arguments are copied, just so the function can work with them without affecting the originals.

The C language, on the other hand, uses the *call by value* scheme in which values, not references, are passed to functions. Within a called function, references to formal arguments see values on the stack, instead of the objects from which they were taken.

The most important point to remember about passing arguments in C is that there is no connection between an actual argument and its source. Changes to the arguments received by a function have no effect whatsoever on the objects that might have supplied their values. They can be changed with abandon and their sources will not be affected in any way. This removes a burden of concern from

the programmer since arguments may be used as local variables without side effects. It also avoids the need to define temporary variables just to prevent side effects.

It is precisely because C uses call by value that we can pass expressions—not just variables—as arguments. The value of an expression can be copied, but it cannot be referenced since it has no existence in memory—it is not an lvalue. Therefore, call by value adds important generality to the language.

Although the C language uses the call by value technique, it is still possible to write functions that have side effects; but it must be done deliberately. This is possible because of C's ability to handle expressions that yield addresses. And, since any expression is a valid argument, addresses can be passed to functions.

There are two ways to refer to objects that are pointed to by arguments—by using the *indirection operator* (*) and by *subscripting*. Recall from Chapters 5 and 6 that pointers may be subscripted just like array names.

To see how a function may produce side-effects through a pointer argument, consider:

```
func (p) int *p; {
  *p = 0;
  }
```

Notice that **p** is declared to be a pointer to integers. The assignment statement does not assign zero to **p** but to the *object at* **p;** that is, the integer to which **p** points.

Since expressions may include *assignment, increment,* and *decrement* operators (see Chapter 9), it is possible for argument expressions to affect the values of arguments lying to their right (recall that Small C evaluates argument expressions from left to right). Consider, for example,

```
func (y=x+1, 2*y);
```

where the first argument has the value **x+1,** and the second argument has the value **2*(x+1).** What would be the value of the second argument if arguments were evaluated from right to left? This kind of situation should be avoided, since

the C language does not guarantee the order of argument evaluation. The safe way to write this is:

```
y=x+1;
func (y, 2*y);
```

As we noted earlier, the Small C compiler does not verify the number and type of arguments passed to functions. It is the programmer's responsibility to ensure that they match the formal arguments in the function's definition. If too many arguments are given, the function will see only the trailing arguments. If too few are given, however, the function will use items on the stack, below the argument list, as though they were arguments; and that will surely produce undesirable results. This mistake—mismatching actual and formal arguments—is perhaps the most common and most troublesome error made by C programmers.

Occasionally, the need arises to write functions that work with a variable number of arguments. An example is **printf**() in the library.

In Full C, these situations are handled by designing such functions so that the first (left-most) argument indicates how many other arguments are being passed. Then, using the knowledge that arguments are pushed onto the stack from right to left (opposite to Small C), the function declares only one argument—the first one written, the last one pushed. The address of that argument is easily assigned to a pointer by writing an address operator (**&**) before the argument name. It then becomes a simple matter to obtain the following arguments by incrementing the pointer. And, since the first argument indicates how many others follow, the function knows when to quit.

However, since Small C pushes arguments from left to right, this technique would only work if such functions were designed to have the last (right-most) argument indicate how many arguments precede it. But that option is not open if functions like **printf**() are to be compatible with their Full C counterparts.

Consequently, Small C provides a means by which a function may determine how many arguments were actually passed to it. With each call, a count of the arguments being passed is placed in the 8086's CL register. This count may be

obtained by calling a special function **CCARGC()** and assigning the returned value to a variable. This function exists in the **CALL** module of the Small C library. **CCARGC()** must be called first in the function since the CPU registers are volatile and the argument count would soon be lost. A statement like:

```
count = CCARGC();
```

will do the job. Four functions in the Small C library require argument counts; they are **printf(), fprintf(), scanf(),** and **fscanf().**

The Stack Frame

The machine stack is used by Small C programs for three purposes—for passing arguments, for saving return addresses, and for allocating local objects. Notice that each of these relates to the invocation of a function. The part of the stack that is used when a function is invoked is called the function's *stack frame.* Chapter 1 pointed out that C programs are initially given control by performing a normal call to **main().** Therefore, every function, including **main(),** operates within the context of a stack frame. Figure 8-1 illustrates the structure of a Small C stack frame.

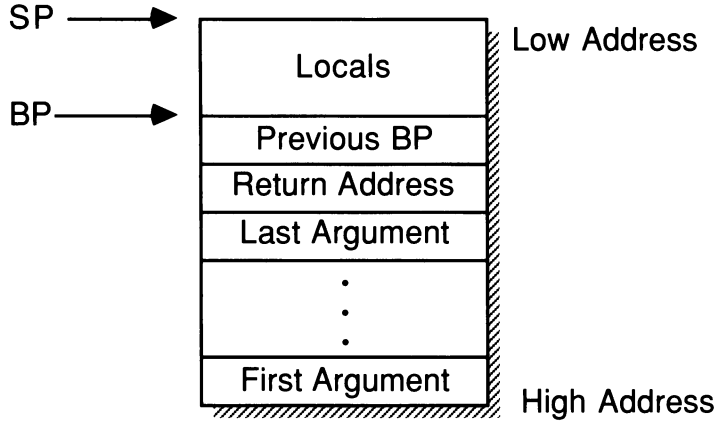


Figure 8-1. The Stack Frame

When a function is to be called, the actual arguments are pushed onto the stack in the order in which they are evaluated—left to right. Then the function is called by means of a **CALL** instruction, which pushes the return address—the address of the following instruction—onto the stack above the arguments.

When the called function receives control, it immediately pushes the previous frame pointer, in the BP register, onto the stack to preserve it. It then moves the stack pointer (SP), to BP, thereby establishing a fixed point of reference for the current stack frame. Arguments are referenced with respect to BP by adding a positive offset. (See Appendix B for an overview of the 80x86 CPU registers.)

As local objects are created, they are allocated on the stack by subtracting from SP. They are then referenced like arguments, except with negative offsets. When control exits a block in which locals were allocated, SP is incremented to its original value so the same stack space may be used by other locals.

When control returns to the caller, BP is moved to SP so that a POP instruction can restore BP to its previous value. Then a return instruction pops the return address from the stack into the processor's *instruction pointer* (IP). This effects the return by causing the next instruction to come from the location indicated by the return address.

Finally, on return from the function, the calling function—which pushed the arguments onto the stack—removes them by the most efficient means possible—by adding a constant to the stack pointer, by popping an argument or two into an unused register, or by taking no action if no arguments were passed to the function.

Each call allocates a stack frame. As the calls are nested, the frames are "stacked" one above the other. And, as they unnest, they are "unstacked" in the reverse order. As a result, there is no interference between function calls, even calls to functions that already have frames on the stack.

Since memory is limited, there is a practical limit to the amount of nesting that can be done. It is the programmer's responsibility to determine that his programs do not overflow the stack space that is available. The Small C library (Chapter 12) includes a function **avail(),** which checks for a stack overflow condition.

Returning from Functions

There are two instances where control is caused to return from a function to its caller. When control reaches the right-most brace of the function's body, an implied return is taken. In that case, no return value is specified and so none is provided. The caller will see whatever happens to be in Small C's primary register at the time of the return. It is important to be sure that any function which is supposed to return a value does not return by this means.

On the other hand, explicit returns are possible by writing statements of the form:

```
return ExpressionList?;
```

where **return** is a required keyword and ***ExpressionList?*** is an optional list of expressions. If an expression (list) is provided, its value (the value of the last expression in the list) is returned to the caller. But if there is not an expression, then, as with implicit returns, the returned value is unpredictable and should not be used by the caller.

In Full C, functions are declared according to the type of value they return. In Small C, however, functions return only integer values. This is a reasonable simplification when we consider that Small C only supports character and integer variables, and that characters are automatically promoted to integers whenever they are referenced. We are free to write functions that return characters, but the value returned will actually be an integer.

Suppose we should want to have a function return a pointer. How can that be done? Since integers and pointers in Small C have the same size, there is no real problem. We can simply write our return expression to produce an address instead of an integer, but we must be aware that the returned value will be seen as an integer in the expression in which the function is called. In some expressions the difference is immaterial, but in others it is important. Where the difference matters, we can break down the expression into two parts. First, assign the function's value to a pointer of the desired type, and then write the pointer in the expression where the function would have been called. For example, if we want-

ed to assign **5** to the character pointed to by **func()** we would write:

```
cp = func();
*cp = 5;
```

where cp is a character pointer.

Recursive Calls

Using the stack for actual arguments, return addresses, and local objects has the advantage of permitting *recursive* calls. A function is called recursively when it either calls itself directly or it calls other functions which directly or indirectly call it. Use of recursion can simplify many algorithms, but it does drive up the amount of stack space needed and it usually makes the program logic less obvious. Take the function **backward()** in Listing 8-1, for example. This function displays the characters of a string in reverse order. It receives the character string (actually a pointer to the first character) as an argument when it is first called. If the character at that address is *not zero,* it calls itself again passing the address of the next character. This nesting continues until a *zero* character is found, at which point the current instance of **backward()** simply returns to the prior one. Control resumes in that instance at **putchar(),** which writes the current character to the standard output file (see Chapter 12). That instance then returns to the prior one, and so on. The calls continue to unnest until the first instance displays the first character and returns to its caller. As an exercise, you might try writing this function without recursion.

```
backward (string) char string[]; {
  if (*string) {
     backward (string + 1);
     putchar (*string);
     }
  }
```

Listing 8-1. Sample Recursive Function Call

A Sample Function

Suppose we need a function which will compare two character strings, returning zero if they differ, and their length if they match. The first string may be a substring of a larger string, so it is not necessarily terminated with a zero byte, whereas the second string (being a full string) does have a null terminator. We might write this function as follows:

```
streq (str1, str2)  char str1[], str2[]; {
  int k;
  k = 0;
  while (str2[k]) {
        if(str1[k] != str2[k]) return 0;
        ++k;
        }
  return k;
}
```

Listing 8-2. String Comparison Function

The function's name is **streq,** meaning *string equality.* It expects two arguments to which it gives the formal names **str1** and **str2.** These are declared to be character arrays; however, since array arguments are passed as addresses, these are actually pointer arguments.

A local integer **k** is defined and initially set to zero; it will serve as both a subscript into the strings and the value to return if the strings match. The **while** statement executes repeatedly as long as the character in the second string subscripted by **k** is not zero; i.e., the end of the second string has not been reached. With each iteration, the compound statement is executed. It first compares corresponding characters (subscripted by **k**) in the two strings. If they are not equal, the function quits by returning zero to the caller. However, if they match, **k** is incremented by **++k.** At this point, the compound statement is exhausted, so the **while** condition is checked again, this time with a larger value of **k.**

If this continues until a zero character is found in **str2,** then a match has occurred since none of the preceding characters differed. In that case, **k** is returned since it contains the length of the matched string. This is true because array elements are numbered from zero. Thus, if **str2** contains three characters and a null terminator, then the terminator will be reached when **k** equals three.

This is one of the string matching functions used by Small C; it finds keywords in programs. The first argument is a pointer into a line of code in a program, and the second is a literal string containing the keyword being sought.

CHAPTER 9

Expressions

Most programming languages support the traditional concept of an expression as a combination of constants, variables, array elements, and function calls joined by various *operators* (+, -, etc.) to produce a single numeric value. Each operator is applied to one or two *operands* (the values operated on) to produce a single value which may itself be an operand for another operator.

This idea is generalized in C by including non-traditional data types and a rich set of operators. Pointers, unsubscripted array names, and function names are allowed as operands. And, as Table 9-1 illustrates, a large number of operators are available. All of these operators can be combined in any useful manner in an expression. As a result, C allows the writing of very compact and efficient expressions which at first glance may seem a bit strange.

Another unusual feature of C is that anywhere the syntax calls for an expression, a list of expressions with comma separators may appear. Expression lists are evaluated from left to right with the right-most expression determining the value of the list.

Before looking at the kinds of expressions we can write in C, we will first consider the process of evaluating expressions and some general properties of operators.

Operator Properties

The basic problem in evaluating expressions is deciding which parts of an expression are to be associated with which operators. To eliminate ambiguity, operators are given three properties: *operand count, precedence,* and *associativity.*

Operand count refers to the classification of operators as *unary, binary,* or *ternary* according to whether they operate on one, two, or three operands. The unary minus sign, for instance, reverses the sign of the following operand, whereas the binary minus sign subtracts one operand from another.

( )	function call	→	= =	equal	→
[ ]	subscript		!=	not equal	
!	logical NOT	←	&	bitwise AND	→
~	one's complement				
++	increment		^	bitwise exclusive OR	→
– –	decrement				
–	unary minus			bitwise inclusive OR	→
*	indirection				
&	address		&&	logical AND	→
sizeof( )	size of				
			||	logical OR	→
*	multiplication	→			
/	division		?:	conditional operator	←
%	modulo (remainder)				
			=	assignment	←
+	addition	→	+=	add and assign	
–	subtraction		– =	subtract and assign	
			*=	multiply and assign	
<<	shift left	→	/=	divide and assign	
>>	shift right		%=	modulo and assign	
			&=	bitwise AND and assign	
<	less than	→	| =	bitwise OR and assign	
<=	less than or equal		^ =	bitwise XOR and assign	
>	greater than		<<=	left shift and assign	
>=	greater than or equal		>>=	right shift and assign	

Table 9-1. Small C Operators

Precedence refers to the priority used in deciding how to associate operands with operators. For instance, the expression:

```
a + b * c
```

would be evaluated by first taking the product of **b** and **c** and then adding **a** to the result. That order is followed because multiplication has a higher precedence than addition.

Parentheses may be used to alter the default precedence or to make it more explicit. They must always be used in matching pairs. Evaluation of parenthesized expressions begins with the innermost parentheses and proceeds outward, with each parenthesized group yielding a single operand. Within each pair of parentheses, evaluation follows the order dictated by operator precedence and subordinate parentheses.

Writing the above example as:

```
(a + b) * c
```

overrides the default precedence, so that the addition is performed before the multiplication. On the other hand,

```
a + (b * c)
```

simply makes the default precedence explicit.

Table 9-1 lists the Small C operators in descending order of precedence. All operators listed together have the same precedence. Except as grouped by parentheses, they are evaluated as they are encountered.

The last property, *associativity,* determines whether evaluation of a succession of operators at a given precedence level proceeds from left to right or from right to left. This is also called *grouping* because the evaluation of each operator/operand(s) group yields a single value which then becomes an operand for the next operator. Of course, the result of the last operation becomes the value of the whole expression. Grouping is shown in Table 9-1 by the arrows in the upper right-hand corner of each precedence box.

Since addition groups left to right, the expression:

```
a + b + c
```

is evaluated by first adding **a** to **b,** then adding **c** to the result.

Operands

Operands that are not the immediate result of the application of an operator are called *primary* operands, or just *primaries.* Examples are constants, variable names, and so on. Although we treat function calling and subscripting as operations, the values they produce are still considered primaries. We might consider these as operations that help fetch primary values, rather than operations that determine values.

As Table 9-1 indicates, a pair of parentheses are considered an operator when they designate a function call; but when they direct the evaluation precedence they do not function as operators. Instead, they direct the expression analyzer to treat the enclosed expression as a simple operand. For this reason, parenthesized expressions are considered to be primaries. When we study the Small C expression analyzer (see Chapter 26) we shall see that precedence-setting parentheses are handled in the same function that deals with primary objects.

Small C accepts twelve kinds of primary operand. They are:

Numeric Constants Numeric constants (decimal, hexadecimal, or octal) are stored internally as 16-bit values. Ordinarily, they are treated as signed integers; however, if they are written without a sign and are greater than 32767 (decimal), Small C treats them as unsigned integers.

Character Constants Chapter 3 described how character constants are stored internally as 16-bit values. Single-character constants always have zeroes in the high-order byte, making them positive. Double-character constants, however, could have the sign bit set if the left-most character is written as an octal escape sequence. Character constants of either type are treated as signed quantities.

String Constants A string constant, in an expression, yields the 16-bit address of the string. Being an address, it is treated as an unsigned quantity.

Integer Variables Integer variables are stored as 16-bit values. They enter into operations either as two's complement signed or as unsigned values, depending on their declarations. Elements of integer arrays are treated as integer variables.

Character Variables As indicated in Chapter 4, character variables are automatically promoted to integers, either by sign extension or zero padding depending on whether they are signed or unsigned. They are treated as two's complement signed or unsigned quantities depending on their declarations. When values are assigned to character variables, the high-order byte is truncated. Elements of character arrays are treated as character variables.

Integer Array Names The unsubscripted name of an integer array yields the address of the array. Being an address it is treated as an unsigned quantity. Furthermore, values added to, or subtracted from, such an address are doubled so the effect of the operation will be to offset the address by the specified number of integers. Also, when the difference of two integer addresses is taken, the result is divided by two so the answer will reflect the number of integers between the addresses.

Character Array Names The unsubscripted name of a character array yields the address of the array. As an address, it is treated as an unsigned quantity. These addresses cause no scaling of values added to or subtracted from them since characters occupy one byte each. Likewise, the difference of two character addresses is not scaled because it already reflects the number of characters between them.

Integer Pointers An integer pointer yields the address of an integer. As an address it is treated as an unsigned quantity. Furthermore, values added to, or subtracted from, such an address are doubled, so the effect of the operation will be to offset the address by the specified number of integers. Moreover, when the

difference of two integer addresses is taken, the result is divided by two so the answer will reflect the number of integers between the addresses.

Character Pointers A character pointer yields the address of a character. As an address, it is treated as an unsigned quantity. Character addresses cause no scaling of values added to or subtracted from them since characters occupy one byte each. Likewise, the difference of two character addresses is not scaled because it already reflects the number of characters between them.

Function Calls Functions in Small C are always considered to yield signed integer values. A function name followed by parentheses generates a call to the function which yields the value returned by the function. If an undefined name is found in an expression, it is implicitly declared to be a function name, and is treated as such.

Function Names A function name without trailing parentheses yields the address of the named function. If an undefined name is encountered in an expression, it is implicitly declared to be a function name, and is treated as such.

Each operator in an expression yields a value which may serve as an operand for another operator. These computed operands inherit certain attributes from their subordinate operands. If either of the subordinate operands of a binary operator is unsigned, then an unsigned operation (if there are distinct signed and unsigned versions of the operation) is performed and the result is considered to be unsigned. If either of the subordinate operands is an address, then the same unsigned treatment is received and the result is also considered to be an address. The operations which have separate signed and unsigned versions are:

* multiply
/ divide
% modulo
> greater than
>= greater than or equal

< less than
<= less than or equal

All other operations are the same whether operating on signed or unsigned quantities.

Constant and Variable Expressions

When the Small C compiler encounters an expression, it does one of two things. If the expression contains only constant (numeric or character) primaries, the result is a constant, so the compiler evaluates the expression and generates the result. However, if the expression contains variables or addresses (e.g., string constants), the result cannot be determined at compile time, so the compiler generates code that will perform the evaluation at run-time.

As we shall see in Chapter 26, the same expression analyzer performs both tasks. It assumes it is dealing with a variable expression, and so generates run-time code in a staging buffer. However, at each step along the way, it knows whether or not it has a constant value. If the entire expression yields a constant, then the code in the buffer is replaced by code that yields the final value directly. This compile-time evaluation of constant expressions means that we can write constants in complex ways, that reveal their constituent parts, without incurring any run-time overhead. This technique can add clarity to a program's logic by documenting the derivation of "magic" numbers.

Operators

Every operator in a Small C expression yields a 16-bit value. Normally, these are numeric values representing numbers, character codes, addresses, or whatever. Some operators, however, produce *Boolean* (logical) values. There are only two such values, *true* and *false*. When an operand is tested for its logical value, zero is interpreted as *false* and any non-zero value is considered *true*. C places no limit on the kinds of values that may be tested logically. It matters not whether a value is a number, an address, or a Boolean value—zero means *false* and non-zero means *true*.

There are four operators which interpret operands as Boolean values: the logical AND (&&), the logical OR (||), the logical NOT (!), and the conditional operator (?:). The first three of these also yield Boolean values. Boolean values produced by operators are always set to one for *true* and zero for *false*.

There are three *bitwise* operators—AND (&), inclusive OR (|), and exclusive OR (^)—which perform logical operations on the individual bits of their operands and reflect the results in the bits of the values they yield. These operators test corresponding bits of two operands to determine the corresponding bit of the result. The same operation is performed simultaneously on all 16 bits of the operands to produce a 16-bit result.

We now look at the individual operators in the order of precedence (high to low) as shown in Table 9-1.

Function Calls and Subscripts

Unless the precedence is changed by parentheses, the following operations are performed before any others. They group left to right.

() Function Call

Parentheses, possibly containing a list of argument expressions, may follow a primary expression. Such parentheses are taken as an operator, causing the function designated by the primary expression to be called. Normally, the primary expression is just a function name. Sometimes it is a pointer reference like **(*fp)**. It might even be a subscripted pointer reference, as in **fpa[5]**. Since Small C does not perform type checking, **fpa** can be an ordinary integer array containing function addresses. When a function name is used, the function is called directly by referencing its label. In other cases, the primary expression is evaluated to produce the function's address in the primary register. The function is then called indirectly through the primary register.

The argument expressions are evaluated from left to right and passed to the called function. The value yielded by this operator is the value returned by the function. In Small C it will always be interpreted as a signed integer.

[ ] Subscript

A pair of square brackets following a primary expression is considered to contain a subscript into an array of objects. The subscript can be any legal expression, and its value will be interpreted as an integer. The primary expression should yield an address. If it produces a character address, then the subscript will be added to it. The resulting address is then used to obtain a character. If the primary expression produces an integer address, then the subscript will be doubled before being added to it. The resulting address is then used to obtain an integer. As we can easily verify, these operations are consistent with the concept of subscripting from zero. The value yielded by this operation is the value of the object referenced.

The primary expression is usually an array name, but pointer names are also frequently used. Since any expression surrounded by parentheses is a primary expression, we can perform arithmetic on array and pointer names before applying the subscript.

Unary Operators The operators in this group operate on a single operand. Normally, they are written to the left of the operand; however, the increment and decrement operators may be written on either side of the operand. All of the operators in this group associate right to left.

! Logical NOT

This operator yields the logical negation of the operand to its right. If the operand is *false,* this operator yields *true* (1). And if this operand is *true,* this operator yields *false* (zero).

~ One's Complement

This operator complements the bits of the operand to its right. Each bit containing 1 becomes zero and each bit containing zero becomes 1.

++ Increment

This operator increments an operand by one. If it precedes the operand, it yields the incremented value. However, if it follows the operand, it yields the

original value in the expression, although it leaves the operand itself incremented. If the operand is an address, it increments to the next character or integer, depending on the type of the address. The operand must be an lvalue.

— Decrement

This operator decrements an operand by one. If it precedes the operand, it yields the decremented value. However, if it follows the operand, it yields the original value in the expression, although it leaves the operand itself decremented. If the operand is an address, it decrements to the prior character or integer depending on the type of the address. The operand must be an lvalue.

- Unary Minus

The unary minus operator reverses the sign of the operand on its right. This is accomplished by taking the two's complement of the operand. This operator is distinguished from the binary minus by its context; in this case there is no operand on the left.

*** Indirection**

When this operator is written to the left of an address it changes the meaning from *address of* to *object at*. In other words, it indirectly obtains the object pointed to by its operand. An integer address yields an integer, and a character address yields a character.

& Address

The address operator yields the address of the operand to its right. In Small C, the address operator does not group; it can only be applied directly to variable, pointer, and subscripted pointer and array names.

sizeof()

To promote the writing of more portable programs, the C language incorporates an unusual operator that yields the size in bytes of a specific object or of objects of a given type. This operator has the form **sizeof(...)** where the ellipsis

stands for a defined object or a data type. Although it looks like a function call, this is really an operator. It has the effect of "returning" an integer constant, which is the size of the named object or of objects of the specified type. If, for example, we have the declaration **int ia[10];** then **sizeof(ia)** will yield the constant 20—the number of bytes occupied by the array.

If a data type, not a specific object, is given, then it is written as a declaration statement without an identifier. For example, **sizeof(unsigned int)** or just **sizeof(unsigned)** yields two since Small C integers occupy two bytes. Likewise, **sizeof(char *)** yields two since that is the size of a character pointer. Small C does not accept the array modifier **[ ]**.

This operator should be used whenever writing code that depends on the size of variables of a given type, or of a specific object. It is legal anywhere a constant is appropriate. And, since it is evaluated at compile time, there is no run-time overhead associated with its use.

Multiplicative Operators The operators in this group associate left to right. Separate signed and unsigned versions of these three operations are implemented. If either operand is unsigned (including addresses), both operands are assumed to be unsigned, an unsigned operation is performed, and the result assumes the unsigned property.

*** Multiplication**

The multiplication operator yields the product of the operands on its left and right. Although the operation produces a 32-bit result, only the lower 16 bits are retained.

/ Division

The division operator yields the quotient of the left operand divided by the right operand.

% Modulo

The modulo operator yields the remainder of the left operand divided by the right operand.

Additive Operators The operators in this group associate left to right.

+ Addition

The addition operator performs an algebraic addition of the two adjoining operands. If one of the operands is an address, the other is interpreted as a character or integer offset, depending on the type of the address. In that case, if the address points to integers, the non-address operand is doubled before the operation is performed. The value generated by an address offset addition is an address of the same type.

- Subtraction

The subtraction operator subtracts the right operand from the left operand. If one of the operands is an address, the other operand is interpreted as a character or integer offset, depending on the type of address. In that case, if the address points to integers, the non-address operand is doubled before the operation is performed. The value generated by an address offset subtraction is an address of the same type.

If both operands are addresses of the same type, the result is adjusted to represent the number of objects lying between them. In the case of integer addresses, the result is divided by two. In the case of character addresses, no adjustment is necessary. The result is an integer, not an address. The final result is likely to be useless under any of the following conditions: (1) both addresses are not of the same type; (2) the first address is smaller than the second; or (3) both addresses are not in the same array.

Shift Operators The operators in this group associate left to right. They yield a value which is the left operand arithmetically shifted left or right the number of bit positions indicated by the right operand.

<< Shift Left

This operator shifts the left operand left the number of bit positions indicated by the right operand. Zeroes are inserted into the low end of the result. For each

position shifted, the left operand is effectively multiplied by two. Thus, **x << 2** multiplies **x** by four. Since shifting is much faster than multiplying, this is the preferred way to multiply by powers of two.

>> Shift Right

This operator shifts the left operand right the number of bit positions indicated by the right operand. The sign bit retains its value and is replicated into the next lower bit with each shift—the sign of the left operand is preserved. For each position shifted, the left operand is effectively divided by two. Thus, **x >> 2** divides **x** by four. Since shifting is much faster than dividing, this is the preferred way to divide by powers of two.

Relational Operators All four of the relational operators perform a comparison of two operands and yield one of the logical values *true* (one) or *false* (zero) according to whether or not the specified relationship is true. If neither of the operands is an address or an unsigned value, a signed comparison is performed. However, if either operand is an address or an unsigned value, then both are considered to be unsigned. These operators associate from left to right.

< Less Than

This operator yields *true* if the left operand is less than the right operand.

<= Less Than or Equal

This operator yields *true* if the left operand is less than or equal to the right operand.

> Greater Than

This operator yields *true* if the left operand is greater than the right operand.

>= Greater Than or Equal

This operator yields *true* if the left operand is greater than or equal to the right operand.

Equality Operators These two operators determine whether or not the two adjoining operands are equal. They return the Boolean value *true* if the specified relationship is *true*, and *false* otherwise. They associate from left to right.

== Equal

This operator yields *true* if the two operands are equal.

!= Not Equal

This operator yields *true* if the two operands are not equal.

Bitwise AND Operator

& Bitwise AND

This operator yields the bitwise AND of the adjoining operands. If both corresponding bits are 1, the corresponding bit of the result is set to 1; however, if either is zero the resulting bit is reset to zero. This operator associates left to right.

Bitwise Exclusive OR Operator

^ Bitwise Exclusive OR

This operator yields the bitwise exclusive OR of the adjoining operands. If either, but not both, corresponding bits are 1, the corresponding bit of the result is set to 1. On the other hand, if both bits are the same (zero or 1) the resulting bit is set to zero. This operation is *exclusive* in the sense that it excludes from the OR the case where both corresponding bits are ones. This operator associates left to right.

Bitwise Inclusive OR Operator

| Bitwise Inclusive OR

This operator yields the bitwise OR of adjacent operands. If either corresponding bit is 1, the corresponding bit of the result is set to one; however, if

both are zero, the resulting bit is zero. It is *inclusive* in the sense that it includes in the OR, the case where both corresponding bits are ones. This operator associates left to right.

Logical AND Operator

&& Logical AND

This operator yields the logical AND of the adjoining operands. If both operands are *true* it yields *true* (one); otherwise, it yields *false* (zero). This operator associates left to right.

If an expression contains a series of these operators at the same precedence level, they are evaluated left to right until the first *false* operand is reached. At that point the outcome is known, so *false* is generated for the entire series, and the right-most operands are not evaluated. This is a standard feature of the C language, so it can be used to advantage without fear of portability problems. For greater efficiency, compound tests should be written with the most frequently *false* operands first; in that case the trailing operands will be evaluated only rarely. This feature, together with the fact that the operands can be expressions of any complexity (including function calls), makes it possible to write very compact code. For example,

```
func1() && func2() && func3() ;
```

is equivalent to:

```
if(func1()) {
  if(func2()) {
  func3() ;
  }
}
```

Also, trailing operands may safely reference variables (e.g., subscripts) which have been verified in preceding tests. For example,

```
if(ptr && *ptr == '-') ...;
```

performs . . . only if the pointer is not null and it points to a hyphen.

Logical OR Operator

|| Logical OR

This operator yields the logical OR of the adjoining operands. If either is *true* it yields *true,* (1); otherwise, it yields *false* (zero). This operator associates left to right.

If the expression contains a series of these operators, the operands are evaluated left to right until the first *true* operand is reached. At that point the outcome is known, so *true* is generated for the entire series, and the right-most operands are not evaluated. This is a standard feature of the C language, so it can be used to advantage without fear of portability problems. For greater efficiency, compound tests should be written with the most frequently *true* operands first; in that case the trailing operands will be evaluated only rarely. This feature, together with the fact that the operands can be expressions of any complexity (including function calls), makes it possible to write very compact code. For example,

```
func1() || func2() || func3() ;
```

is equivalent to:

```
if(func1()) ;
else if(func2()) ;
  else func3();
```

Conditional Operator

?: Conditional Operator

This unusual operator tests the Boolean value of one expression to determine which of two other expressions to evaluate. Since it is the only operator that requires three operands it is sometimes called the *ternary* operator. It consists of two characters—a question mark and a colon—which separate the three expressions. The general form is

```
Expression1 ? Expression2 : Expression3
```

This sequence may appear anywhere an expression (or subexpression) is acceptable. ***Expression1*** is evaluated for *true* (non-zero) or *false* (zero). If it yields *true* then ***Expression2*** is evaluated to determine the value of the operator. However, if ***Expression1*** is *false* then ***Expression3*** determines the result. Only one of the last two expressions is evaluated. Consider this example:

```
(y ? x/y : x) + 5
```

If **y** is not zero, this expression yields **(x/y)+5;** otherwise, **x+5.**

Unless they are enclosed in parentheses, there are restrictions on the operators that can be used in the three expressions. The first expression can contain only operators that are higher in precedence than the conditional operator; that is, assignment and conditional operators are disallowed. The second and third expressions cannot contain assignment operators, but may have conditional operators, or any operators of higher precedence. Considering these restrictions, and the fact that this operator associates from right to left, the expression:

```
a ? b : c ? d : e
```

is equivalent to:

```
a ? b : (c ? d : e)
```

And,

```
a ? b ? c : d : e
```

is equivalent to:

```
a ? (b ? c : d) : e
```

Obviously, whenever more than one conditional operator is used in combination, it is a good idea to use parentheses even if they are not strictly needed.

Since this operator must yield a single value, and that value comes from either of two expressions which could yield different data types, some further restrictions must apply to the second and third expressions. Otherwise, the compiler might not be able to assign a data type to the result. In other words, it is impossible to determine the attributes of the result at compile time if they are allowed to be determined at run-time. Therefore, the last two expressions must observe the following additional restrictions:

1. Both expressions yield numeric constants. In this case, because the two possible constant values are assumed to be different (otherwise, there would be no point selecting between them), the result is considered to be an integer variable.

2. Either one or the other yields a numeric constant. In these cases, the numeric constant (often zero) is considered to be an exceptional value of the same type as the other expression. Therefore, the result is given the attributes of the non-constant expression.

3. Both expressions yield addresses of the same data type. In this case, both expressions have the same properties, and naturally the result is the same.

4. Both yield integer values. Likewise, in this case, the expression properties match and, therefore, determine the resulting attributes.

If none of these conditions is met, then the message, **"mismatched expressions"** is issued.

Assignment Operators Each of these operators assigns a value to the object on its left side. These operators associate right to left. The new value is either the right-hand operand or a value derived from both operands.

The left (receiving) operand must be an lvalue—it must be an object which occupies space in memory, and is therefore capable of being changed. An unadorned array name, **array** for instance, will not do; whereas, ***array** and **array[x]** are lvalues. Other operands that do not qualify as lvalues are calculated values which are not prefixed by the indirection operator, and names which are prefixed by the address operator. Conversely, the only allowable lvalues in Small C are (1) variable names, (2) pointer names, (3) subscripted pointer names, (4) subscripted array names, and (5) subexpressions that are prefixed by the indirection operator.

If the receiving object is a character, it receives only the low-order byte of the value being assigned.

All but one of the assignment operators have the form **?=**, where **?** stands for a specific non-assignment operator. These operators provide a shorthand way of writing expressions of the form:

```
a = a ? b
```

Thus,

```
a += b
```

is equivalent to:

```
a = a + b
```

And,

```
a <<= b
```

is equivalent to:

```
a = a << b
```

and so on. These operators produce more efficient code, since the receiving operand is evaluated only once.

UNIX C originally implemented these operators with the equal sign leading instead of trailing. The result was that some of them (**=+, =-, =***, and **=&**) were ambiguous. Small C always takes these as a pair of separate operators. To avoid portability problems, always put a space between these character pairs.

Since assignment is a part of expression evaluation, traditional assignment statements are really just stand-alone expressions. And, since all operators yield values which may be used in the further process of expression evaluation, any number of assignment operators may appear in an expression. Most common are multiple assignments like:

```
a = b = c = 5
```

Since these operators group right to left, **5** is first assigned to **c,** then the right-most operator yields the value assigned. The middle operator then assigns that value to **b** and yields the same value for the next operator. Finally, the left-most operator assigns that value to **a.** It is the same as if the expression were written:

```
a = (b = (c = 5))
```

Expressions like:

```
val[i] = i = 5
```

deserve special consideration. Notice that the variable **i** is modified on the right side of the first assignment operator and is also used as a subscript on the left side. Small C uses the original value of **i** for the subscript, but full C uses the modified value. It is best to avoid such expressions to maintain compatibility with Full C.

The assignment operators are:

= Assignment

This operator assigns the value of the right operand to the left operand.

+= Add and Assign

This operator assigns the sum of the left and right operands to the left operand.

-= Subtract and Assign

This operator subtracts the right operand from the left operand and assigns the result to the left operand.

***= Multiply and Assign**

This operator assigns the product of the left and right operands to the left operand.

/= Divide and Assign

This operator divides the left operand by the right operand and assigns the quotient to the left operand.

%= Modulo and Assign

This operator divides the left operand by the right operand and assigns the remainder to the left operand.

&= Bitwise AND and Assign

This operator assigns to the left operand the bitwise AND of the left and right operands.

|= Bitwise Inclusive OR and Assign

This operator assigns to the left operand the bitwise inclusive OR of the left and right operands.

^= Bitwise Exclusive OR and Assign

This operator assigns to the left operand the bitwise exclusive OR of the left and right operands.

<<= Left Shift and Assign

This operator shifts the left operand left the number of bits indicated by the right operand. The result is assigned to the left operand.

>>= Right Shift and Assign

This operator shifts the left operand right the number of bits indicated by the right operand. The result is assigned to the left operand.

CHAPTER 10

Statements

Every procedural language provides *statements* for determining the flow of control within programs. Although declarations are a type of statement, in C the unqualified word *statement* usually refers to procedural statements rather than declarations. In this chapter we are concerned only with procedural statements.

In the C language, statements can be written only within the body of a function; more specifically, only within compound statements. The normal flow of control among statements is sequential, proceeding from one statement to the next. However, as we shall see, most of the statements in C are designed to alter this sequential flow so that algorithms of arbitrary complexity can be implemented. This is done with statements that control whether or not other statements execute and, if so, how many times. Furthermore, the ability to write compound statements permits the writing of a sequence of statements wherever a single, possibly controlled, statement is allowed. These two features provide the necessary generality to implement any algorithm, and to do it in a structured way.

The C language uses semicolons as statement terminators. A semicolon follows every simple (non-compound) statement, even the last one in a sequence.

When one statement controls other statements, a terminator is applied only to the controlled statements. Thus we would write

```
if(x > 5) x = 0; else ++x;
```

with two semicolons, not three. Perhaps one good way to remember this is to think of statements that control other statements as "super" statements that "contain" ordinary (simple and compound) statements. Then remember that only simple statements are terminated. This implies, as stated above, that compound

statements are not terminated with semicolons. Thus,

```
while(x < 5) {func(); ++x;}
```

is perfectly correct. Notice, however, that each of the simple statements within the compound statement is terminated.

Null Statements

The simplest C statement is the null statement. It has no text, just a semicolon terminator. As its name implies, it does exactly nothing. Why have a statement that serves no purpose? Well, as it turns out, statements that do nothing can serve a purpose. As we saw in Chapter 9, expressions in C can do work beyond that of simply yielding a value. In fact, in C programs, all of the work is accomplished by expressions; this includes assignments and calls to functions that invoke operating system services such as input/output operations. It follows that anything can be done at any point in the syntax that calls for an expression. Take, for example, the statement:

```
while(...) ;
```

in which the ellipsis stands for some expression that controls the execution of the null statement following. We are free to write ... so that it produces side effects of any kind in addition to controlling the loop. In other words, the effects of the loop can be accomplished in the controlling expression rather than the controlled statement. The result of this practice is more efficient object code. For a specific example, see the discussion of the **while** statement below. The null statement is just one way in which the C language follows a philosophy of attaching intuitive meanings to seemingly incomplete constructs. The idea is to make the language as general as possible by having the least number of disallowed constructs.

Compound Statements

The terms *compound statement* and *block* both refer to a collection of statements that are enclosed in braces to form a single unit. Compound statements have the form

```
{ObjectDeclaration?... Statement?...}
```

ObjectDeclaration?... is an optional set of local declarations. If present, Small C requires that they precede the statements; in other words, they must be written at the head of the block.

Statement?... is a series of zero or more simple or compound statements. Notice that there is not a semicolon at the end of a block; the closing brace suffices to delimit the end.

The power of compound statements derives from the fact that one may be placed anywhere the syntax calls for a statement. Thus, any statement that controls other statements is able to control units of logic of any complexity.

When control passes into a compound statement, two things happen. First, space is reserved on the stack for the storage of local variables that are declared at the head of the block. Then the executable statements are processed.

When control leaves a compound statement, the stack is restored to its original condition. For more on local declarations, see chapters 4–6.

One important limitation of Small C is that a block containing local declarations must be entered through its leading brace. This is because bypassing the head of a block effectively skips the logic that reserves space for local objects. Since the **goto** and **switch** statements (below) could violate this rule, Small C mutually excludes from any function **goto** statements and declarations at levels below the function body, and it disallows declarations within **switch** statements. Full C has no such restrictions, however. As it turns out, these restrictions are seldom encountered, since most people declare all of their locals at the head of function bodies, and it is fairly unusual to see locals declared in blocks within **switch** statements.

Expression Statements

The C language is unusual in its provision for *expression statements*—expressions which the compiler accepts as complete statements. We saw in Chapter 8 that this generalization leads to the elimination of procedures as a distinct class of subroutine and the elimination of special call statements for invoking them. This feature and the provision of assignment operators also leads to the elimination of a special assignment statement.

The value produced by an expression statement is not used under normal conditions. In fact, some compilers optimize such values into oblivion. But Small C goes ahead and produces them in its primary register and then ignores them. It is possible, however, to utilize these values by inserting assembly language code after the expression statement (see Chapter 11 for details).

As we noted above, the work in C programs is really done by expressions. Function calls, assignments, increments, and decrements are all performed during expression evaluation. That being the case, it follows that C must provide for the normal sequential evaluation of expressions; it must support expression statements. Some examples of valid expression statements are:

```
func();
++i;
i = 15, j = 16, k = 17;
x += func(x, y = 12) * 100;
```

The Goto Statement

Goto statements break the sequential flow of execution by causing control to jump abruptly to designated points. They have the general form:

```
goto Name;
```

where ***Name*** is the name of a label which must appear in the same function. It must also be unique within the function. Labels have the form:

```
Name:
```

where ***Name*** obeys the C naming conventions (see Chapter 2). Notice that labels are terminated with a colon. This highlights the fact that they are not statements but statement prefixes which serve to label points in the logic as targets for **goto** statements. When control reaches a **goto,** it proceeds directly from there to the designated label. Both forward and backward references are allowed, but the range of the jump is limited to the body of the function containing the **goto** statement.

As we observed above, **goto** statements, cannot be used in functions which declare locals in blocks which are subordinate to the outermost block of the function.

Goto statements should be used sparingly, if at all. Overreliance on them is a sign of sloppy thinking. Strictly speaking, they are never absolutely necessary. In fact, the original Small C compiler did not even support **goto** statements. But situations do occasionally arise in which a **goto** can save the writing of redundant statements. To keep the meaning of your logic clear, you should try to limit the distance between **goto** statements and their target labels. It also helps to write labels so they stand out from their surrounding code; for instance, placing a blank line above a label helps.

The If Statement

If statements provide a non-iterative choice between alternate paths based on specified conditions. They have either of two forms:

```
if (ExpressionList) Statement
```

or:

```
if (ExpressionList) Statement else Statement
```

ExpressionList is a list of one or more expressions, and ***Statement*** is any simple or compound statement. First, ***ExpressionList*** is evaluated and tested. If more than one expression is given, they are evaluated from left to right and the right-most expression is tested. If the result is *true* (non-zero), then the first (or

only) statement is executed and the statement following the keyword **else** (if present) is skipped. If it is *false* (zero), the first statement is skipped and the second statement (if present) is executed. For example, the statement:

```
if (ch) {
  putchar(ch);
  ++i;
  }
else return (i);
```

tests **ch.** If it is not zero, **putchar()** (see Chapter 12) writes it to the standard output file and i is incremented; otherwise, the value of i is returned to the calling function.

The Switch Statement

Switch statements provide a non-iterative choice between any number of paths based on specified conditions. They compare an expression to a set of constant values. Selected statements are then executed depending on which value, if any, matches the expression. **Switch** statements have the form:

```
switch (ExpressionList) {Statement?...}
```

where ***ExpressionList*** is a list of one or more expressions. ***Statement?...*** represents the statements to be selected for execution. They are selected by means of **case** and **default** prefixes—special labels that are used only within **switch** statements. These prefixes locate points to which control jumps depending on the value of ***ExpressionList.*** They are to the **switch** statement what ordinary labels are to the **goto** statement. They may occur only within the braces that delimit the body of a **switch** statement.

The **case** prefix has the form

```
case ConstantExpression :
```

The **default** prefix has the form

```
default:
```

The terminating colons are required; they heighten the analogy to ordinary statement labels. Any expression involving only numeric and character constants and operators is valid in the **case** prefix.

After evaluating ***ExpressionList,*** a search is made for the first matching **case** prefix. Control then goes directly to that point and proceeds normally from there. Other **case** prefixes and the **default** prefix have no effect once a case has been selected; control flows through them just as though they were not even there. If no matching **case** is found, control goes to the **default** prefix, if there is one. In the absence of a **default** prefix, the entire compound statement is ignored, and control resumes with whatever follows the **switch** statement. Only one **default** prefix may be used with each **switch.** Full C compilers reject multiple **case** prefixes with the same value. Small C accepts them, but effectively sees only the first one.

If it is not desirable to have control proceed from the selected prefix all the way to the end of the **switch** block, **break** statements may be used to exit the block. **Break** statements have the form:

```
break;
```

Some examples may help clarify these ideas. The statement:

```
switch (ch) {
  case 'Y':
  case 'y': cptr = "yes";
            break;
  case 'N':
  case 'n': cptr = "no";
            break;
   default: cptr = "error";
}
```

tests **ch.** If it equals **'Y'** or **'y'** the character pointer **cptr** is set to the address of a string containing **"yes"** and control exits the **switch** block. If it equals **'N'** or **'n'** then **cptr** is set to the address of a string containing **"no"** and control exits from the block. Those cases failing, **cptr** is set to the address of a string containing **"error"** and control exits through the end of the block.

The statement:

```
switch (i) {
  default: putchar ('a');
   case 2: putchar ('b');
   case 3: putchar ('c');
  }
```

tests **i** for values **2** and **3.** Those failing, the characters **abc** are written to the standard output file. If **i** is **2** then just **bc** is written; and if it is **3** then only **c** is written.

The body of the **switch** is not a normal compound statement since local declarations are not allowed in it or in subordinate blocks. This restriction enforces the Small C rule that a block containing declarations must be entered through its leading brace.

The While Statement

The **while** statement is one of three statements that determine the repeated execution of a controlled statement. This statement alone is sufficient for all loop control needs. In fact, it was the only such statement supported by the original Small C compiler. The other two merely provide an improved syntax and an execute-first feature. **While** statements have the form:

```
while ( ExpressionList ) Statement
```

where ***ExpressionList*** is a list of one or more expressions, and ***Statement*** is a simple or compound statement. If more than one expression is given, the rightmost expression yields the value to be tested. First, ***ExpressionList*** is evaluated.

If it yields *true* (non-zero), then ***Statement*** is executed and ***ExpressionList*** is evaluated again. As long as it yields *true*, ***Statement*** executes repeatedly. When it yields *false*, ***Statement*** is skipped, and control continues with whatever follows. In the example

```
i = 5;
while (i) array[-i] = 0;
```

elements 0 through 4 of **array** are set to zero. First **i** is set to **5.** Then as long as it is not zero, the assignment statement is executed. With each execution **i** is decremented before being used as a subscript.

It is common to see **while** statements where the statement being controlled is null and all of the work is done in the expression being tested. In the statement:

```
while (*dest++ = *sour++) ;
```

dest and **sour** are pointers. With every iteration, the *object at* **sour** is obtained (before incrementing **sour**) and is then assigned to the *object at* **dest** (before incrementing **dest**). Since the assignment operator yields the value assigned, that becomes the value of the expression in parentheses. If it is not zero, the null statement executes (doing nothing), and another evaluation is performed. The process repeats until a value of zero has been assigned. This illustrates how strings in C are usually copied.

Two other statements are handy for use with the **while** (or any loop controlling) statement when it controls a compound statement. The **continue** statement has the form:

```
continue;
```

It causes control to jump directly back to the top of the loop for the next evaluation of the controlling expression. If loop controlling statements are nested, then **continue** affects only the innermost surrounding statement. That is, the innermost loop statement containing the **continue** is the one that starts its next iteration.

The **break** statement (described earlier) may also be used to break out of loops. It causes control to pass on to whatever follows the loop controlling statement. If **while** (or any loop or **switch**) statements are nested, then **break** affects only the innermost statement containing the **break.** That is, it exits only one level of nesting.

It is not uncommon to see **while** statements such as:

```
while (1) {
  ...
  if(...) break;
  ...
  }
```

Notice that the expression is always *true*. When a specific condition arises, control breaks out of the loop. This is the usual way out of such a loop. Of course, the program could use a **goto** or a **return** statement (below), or it could terminate execution by calling the **exit()** or **abort()** (see Chapter 12).

The For Statement

The **for** statement also controls loops. It is really just an embellished **while** in which the three operations normally performed on loop-control variables (initialize, test, and modify) are brought together syntactically. It has the form

```
for (ExpressionList?;
     ExpressionList?;
     ExpressionList?) Statement
```

For statements are performed in the following steps:

1. The first ***ExpressionList*** is evaluated. This is done only once to initialize the control variable(s).

2. The second ***ExpressionList*** is evaluated to determine whether or not to perform ***Statement.*** If more than one expression is given, the right-most expres-

sion yields the value to be tested. If it yields *false* (zero), control passes on to whatever follows the **for** statement. But, if it yields *true* (non-zero), ***Statement*** executes.

3. The third ***ExpressionList*** is then evaluated to adjust the control variable(s) for the next pass, and the process goes back to step 2.

The example given previously in which a five-element array is set to zero could be written as:

```
for (i = 4; i >= 0; -i) array[i] = 0;
```

or a little more efficiently as:

```
for (i = 5; i; array[-i] = 0) ;
```

Any of the three expression lists may be omitted, but the semicolon separators must be kept. If the test expression is absent, the result is always *true*. Thus,

```
for (;;) {...break;...}
```

will execute until the **break** is encountered. This syntax is not as desirable as **while(1)...,** however, since Small C implements **for** statements with more hidden jumps than **while** statements (see Chapter 19).

As with the **while** statement, **break** and **continue** statements may be used with equivalent effects. A **break** statement makes control jump directly to whatever follows the **for** statement. And a **continue** skips whatever remains in the controlled block so that the third ***ExpressionList*** is evaluated, after which the second one is evaluated and tested. In other words, a **continue** has the same effect as transferring control directly to the end of the block controlled by the **for.**

The Do Statement

The **do** statement is the third loop controlling statement in C. It is really just an execute-first **while** statement. It has the form:

```
do Statement while (ExpressionList);
```

Statement is any simple or compound statement. The **do** statement executes in the following steps:

1. ***Statement*** is executed.
2. Then, ***ExpressionList*** is evaluated and tested. If more than one expression is given, the right-most expression yields the value to be tested. If it yields *true* (non-zero), control goes back to step 1; otherwise, it goes on to whatever follows.

As with the **while** and **for** statements, **break** and **continue** statements may be used. In this case, a **continue** causes control to proceed directly down to the **while** part of the statement for another test of ***ExpressionList.*** A **break** makes control exit to whatever follows the **do** statement.

The example of the five-element array could be written as:

```
i = 4;
do {array[i] = 0; -i;} while (i >= 0);
```

or as:

```
i = 4;
do array[i--] = 0; while (i >= 0);
```

or as:

```
i = 5;
do array[--i] = 0; while (i);
```

The Return Statement

The **return** statement is used within a function to return control to the caller. **Return** statements are not always required, since reaching the end of a function

always implies a return. But they are required when it becomes necessary to return from interior points within a function or when a useful value is to be returned to the caller. **Return** statements have the form:

```
return ExpressionList?;
```

ExpressionList? is an optional list of expressions. If present, the last expression determines the value to be returned by the function. If absent, the returned value is unpredictable.

Missing Statements

It may be surprising that nothing was said about input/output, program control, or memory management statements. The reason is that such statements do not exist in the C language proper.

In the interest of portability, these services have been relegated to a set of standard functions in the run-time library. Since they depend so heavily on the run-time environment, removing them from the language eliminates a major source of compatibility problems. Each implementation of C has its own library of standard functions which perform these operations. Since different compilers have libraries that are pretty much functionally equivalent, programs have very few problems when they are compiled by different compilers. Chapter 12 describes the standard functions of the Small C compiler.

CHAPTER 11

Preprocessor Directives

C compilers incorporate a preprocessing phase that alters the source code in various ways before passing it on for compiling. Four capabilities are provided by this facility in Small C. They are:

1. macro processing
2. inclusion of text from other files
3. conditional compiling
4. in-line assembly language

The preprocessor is controlled by directives which are not part of the C language proper. Each directive begins with a # character and is written on a line by itself. Only the preprocessor sees these directive lines since it deletes them from the code stream after processing them.

Depending on the compiler, the preprocessor may be a separate program or it may be integrated into the compiler itself. Small C has an integrated preprocessor that operates at the front end of its single pass algorithm.

Macro Processing

Directives of the form:

```
#define Name CharacterString?...
```

define names which stand for arbitrary strings of text. After such a definition, the preprocessor replaces each occurrence of **Name** (except in string constants and character constants) in the source text with ***CharacterString?...*** As Small C implements this facility, the term "macro" is misleading, since

parameterized substitutions are not supported. That is, ***CharacterString?...*** does not change from one substitution to another according to parameters provided with ***Name*** in the source text.

Small C accepts macro definitions only at the global level. It should be obvious that the term *definition,* as it relates to macros, does not carry the special meaning it has with declarations (Chapters 4-6).

The Name part of a macro definition must conform to the standard C naming conventions as described in Chapter 2. ***CharacterString?...*** begins with the first printable character following ***Name*** and continues through the last printable character of the line, or until a comment is reached.

If ***CharacterString?...*** is missing, occurrences of ***Name*** are simply squeezed out of the text. Name matching is based on the whole name (up to eight characters); part of a name will not match. Thus the directive:

```
#define ABC 10
```

will change,

```
i = ABC;
```

to:

```
i = 10;
```

but it will have no effect on,

```
i = ABCD;
```

It is customary to use uppercase letters for macro names to distinguish them from variable names.

Replacement is also performed on subsequent **#define** directives, so that new symbols may be defined in terms of preceding ones.

The most common use of **#define** directives is to give meaningful names to constants—to define so-called *manifest* constants. However, we may replace a

name with anything at all, a commonly occurring expression or sequence of statements for instance. Some people are fond of writing:

```
#define FOREVER while(1)
```

and then writing their infinite loops as:

```
FOREVER {...}
```

Conditional Compiling

This preprocessing feature lets us designate parts of a program which may or may not be compiled, depending on whether or not certain symbols have been defined. In this way it is possible to write into a program, optional features which are chosen for inclusion or exclusion by simply adding or removing **#define** directives at the beginning of the program.

When the preprocessor encounters:

```
#ifdef Name
```

it looks to see if the designated name has been defined. If not, it throws away the following source lines until it finds a matching:

```
#else
```

or,

```
#endif
```

directive. The **#endif** directive delimits the section of text controlled by **#ifdef,** and the **#else** directive permits us to split conditional text into *true* and *false* parts. The first part **(#ifdef...#else)** is compiled only if the designated name is defined, and the second **(#else...#endif)** only if it is not defined.

The converse of **#ifdef** is the:

```
#ifndef Name
```

directive. This directive also takes matching **#else** and **#endif** directives. In this case, however, if the designated name is *not* defined, then the first **(#ifndef...#else)** or only **(#ifndef...#endif)** section of text is compiled; otherwise, the second **(#else...#endif),** if present, is compiled.

Nesting of these directives is allowed; and there is no limit on the depth of nesting. It is possible, for instance, to write something like:

```
#ifdef ABC
...    /* ABC */
   #ifndef DEF
...    /* ABC and not DEF */
   #else
...    /* ABC and DEF */
   #endif
...    /* ABC */
#else
...    /* not ABC */
   #ifdef HIJ
...    /* not ABC but HIJ */
   #endif
...    /* not ABC */
#endif
```

where the ellipses represent conditionally compiled code, and the comments indicate the conditions under which the various sections of code are compiled.

Including Other Source Files

The preprocessor also recognizes directives to include source code from other files. The three directives:

```
#include "Filename"
#include <Filename>
#include Filename
```

cause a designated file to be read as input to the compiler. The preprocessor replaces these directives with the contents of the designated files. When the files are exhausted, normal processing resumes.

Filename follows the normal MS-DOS file specification format, including drive, path, filename, and extension. Full C requires either quotation marks or angle brackets around the name, but Small C is not so particular. Nevertheless, for better portability, we should write:

```
#include <stdio.h>
```

to include the standard I/O header file (which contains standard definitions and is normally included in every C program), and for other files:

```
#include "Filename"
```

Use of this directive allows us to draw upon a collection of common functions which can be included into many different programs. This reduces the amount of effort needed to develop programs and promotes uniformity among programs. However, since this method requires the recompiling of the code in each program that uses it, its use is usually limited to including header files of common macro definitions and global declarations. On the other hand, procedural modules are usually compiled separately, stored in libraries, and combined with programs at link time (see Chapter 16).

Assembly Language Code

One of the main reasons for using the C language is to achieve portability. But there are occasional situations in which it is necessary to sacrifice portability in order to gain full access to the operating system or to the hardware in order to perform some interface requirement. If these instances are kept to a minimum and are not replicated in many different programs, the negative effect on portability may be acceptable.

To support this capability, Small C provides for assembly language instructions to be written into C programs anywhere a statement is valid. Since the compiler generates assembly language as output, when it encounters assembly language instructions in the input, it simply copies them directly to the output.

Two special directives delimit assembly language code. They are:

```
#asm
```

and,

```
#endasm
```

Everything from **#asm** to **#endasm** is assumed to be assembly language code and so is sent straight to the output of the compiler exactly as it appears in the input. Macro substitution is not performed.

Of course, to make use of this feature, we must know how the compiler uses the CPU registers, how functions are called, and how the operating system and hardware works. Small C code generation is covered in Chapter 19.

PART 2

THE SMALL C COMPILER

PART 2

The Small C Compiler

Part 1 only dealt with the Small C language; nothing pertaining to the practical aspects of program development was covered. Here, in Part 2, we deal with the practical aspects of using the Small C compiler as a program development tool. Specifically, we will cover input/output, program control, memory management, and the other facilities of the Small C library. Also, we will discuss invoking the compiler and the assembly and link steps, and writing efficient or portable code was mentioned.

The following six chapters cover: (1) the Small C library functions, (2) efficiency considerations, (3) compatibility with full C, (4) invoking Small C programs, (5) invoking the compiler, and (6) compiling new versions of the compiler.

CHAPTER 12

Library Functions

This chapter presents the functions in Small C's standard library. These functions exist as object modules which are automatically fetched by the linker when they are needed. All we have to do in our programs is call these functions when we require their services. The compiler automatically declares them to be external, and the linker automatically fetches them from the library and links them to our programs as it builds executable files.

We are free to define in our program's functions that have the same names as library functions. When we do, the defined functions override their library counterparts. The overridden library functions cannot be reached from such programs.

To enhance the portability of programs, C compilers (including Small C) implement a large subset of their library functions in a standard way—with the same names, arguments, return values, and functionality. However, some functions are unique to the particular compiler. Functions which are unique to Small C are identified below as Small C functions. Their use will detract from the portability of programs. However, since the source code for the library is available, we have the option of porting these unique functions together with our programs. In many cases no changes will be required. In other cases alterations may be necessary. And in some cases porting the functions will be out of the question; changing our programs would be easier.

The file **STDIO.H** should be included at the beginning of every program by writing the directive:

```
#include <stdio.h>
```

This header file contains the definitions of a set of standard symbols which are used as file descriptors (below) and as values that are returned by the library functions. They are:

```
#define stdin0        /* fd for standard input file */
#define stdout   1    /* fd for standard output file */
#define stderr   2    /* fd for standard error file */
#define stdaux   3    /* fd for standard comm port (COM1:) */
#define stdprn   4    /* fd for standard printer port (LPT1:) */
#define ERR     -2    /* error condition return value */
#define EOF     -1    /* end of file return value */
#define NULL 0        /* null value */
```

Also included in **STDIO.H** are some miscellaneous definitions which are handy to have around. They are:

```
#define YES          1      /* Boolean true value */
#define NO           0      /* Boolean false value */
#define NULL         0      /* a null pointer or ASCII byte */
#define CR           13     /* ASCII carriage return */
#define LF           10     /* ASCII line feed */
#define BELL         7      /* ASCII bell */
#define SPACE        ' '    /* ASCII space */
#define NEWLINE LF          /* the Small C newline character */
```

Input/Output Functions

The functions in this group give programs access to the outside world through the reading and writing of files. Even the keyboard, the screen, and devices connected to the serial and parallel ports are treated as files.

Small C differs somewhat from Full C in the way it designates which file an I/O function is to reference. Full C compilers refer to I/O functions in two ways. Their high-level I/O functions accept a pointer to a file-control structure as a means of identifying the pertinent file; each open file has such a structure. But their low-level functions employ a small integer value, called a *file descriptor* (**fd**), instead. They have separate high- and low-level open functions that return a pointer or a file descriptor respectively for the newly opened file. The returned

value is retained by the program and used thereafter to reference the file.

Small C, on the other hand, uses file descriptors exclusively. This may seem like a major difference, but in fact it is not very significant. It matters not whether the value returned by an open function is a pointer or integer. In either case, the program simply holds on to it, and passes it back to functions that expect it. When Small C programs are ported to full C compilers, their file descriptors may have to be redeclared as pointers. When both low-level and high-level functions are used, it will be necessary to define pointers in addition to the existing file descriptors. Full C provides functions for mapping between pointers and file descriptors.

Five *standard* files are automatically opened and waiting for use whenever a Small C program begins execution. Their file descriptors have fixed values as defined by the first five *#define* directives above. Table 12-1 also shows their default open modes, and redirection capabilities. These files should be referenced by means of their standard names.

Fd	Name	Default Assignment	Open Mode	Comment
0	`stdin`	keyboard	read	redirectable
1	`stdout`	screen	write	redirectable
2	`stderr`	screen	write	not redirectable
3	`stdaux`	`COM1`:port	read/write	not redirectable
4	`stdprn`	`LPT1`: port	write	not redirectable

Table 12-1. Standard File-Descriptor Assignments

The *standard input* file (**stdin**) is automatically opened for reading and is assigned to the console keyboard by default. However, when the program is invoked, this assignment can be *redirected* from the keyboard to any other input device or file simply by placing a *redirection specification* (see Chapter 15) in the command line. The advantage of this arrangement is obvious: it provides file independence. Programs can be written without reference to a specific input file

or device name. They can then be associated with any given file at run time. If no redirection specification is given, **stdin** reads the keyboard.

The *standard output* file (**stdout**) is automatically opened for writing and is assigned to the console screen by default. This file, too, can be redirected at run time to any other output device or file (see Chapter 15).

The *standard error* file (**stderr**) is also automatically opened for writing and is also assigned to the console screen. It differs from **stdout,** however, in that it cannot be redirected. This file is intended for the writing of error messages. By separating normal program output from error messages, only useful output data gets redirected with **stdout,** while error messages continue to go to the screen. Of course, **stderr** is suitable for all kinds of operator communication, not just error messages.

These are the traditional standard files for C programs. However, since other MS-DOS C compilers implement standard files for the first serial and parallel ports, Small C does too. The *standard auxiliary* file (**stdaux**) is automatically opened for reading and writing, and is assigned to the first serial port (**COM1:**). And the *standard printer* file (**stdprn**) is automatically opened for writing, and is assigned to the first parallel port (**LPT1:**).

Although three of the standard files cannot be redirected, they can be closed and reopened for reading and/or writing with assignment to any device or file—as can any file.

Lest we leave the impression that redirection is the only means of achieving file/device independence in C programs, let us note that it is not. As we shall see momentarily, arguments of any type (even file names) can be conveyed from the command line that invokes a program to the program itself. So there is no reason why file and device names cannot be supplied to programs through this means and used to open files. While this method does require the fetching of command line arguments and the opening of files, it is not limited to just one input and one output file as the redirection mechanism is.

When a file is opened, it receives the lowest unused file descriptor value. If the standard files are not closed, then other files receive file descriptors beginning at 5. Twenty files maximum can be open simultaneously in Small C.

The data transfer functions in the Small C library fall into two groups: there are high-level, *character stream* (ASCII) functions which incorporate the words **get** and **put** in their names; and there are the low-level, *binary* functions which incorporate the words **read** and **write.** These groups of functions differ uniformly with respect to whether or not they process the data which passes through them. This processing is sometimes called "cooking." As we might expect, uncooked data is said to be "raw." The ASCII functions "cook" the data, whereas the binary functions do not. Small C does most of the cooking in the functions **fgetc()** and **fputc().** Since the other **get** and **put** functions call these functions, they also cook their data. The cooking process differs according to whether the data transfer is in or out. On input, the following actions occur:

1. If the file is assigned to the keyboard, the input characters are echoed to the screen. If the character is a carriage return, it is echoed as a two-character sequence: carriage return, and line feed.

2. If the file is assigned to the keyboard and the character is a <control-C>, the program immediately terminates execution with an exit code of 2. This code can be tested in batch files. The <control-C> character does not echo.

3. If the file is assigned to the keyboard and the character is an ASCII DEL **(0x7F)** it is changed to a backspace. Therefore, both the backspace and delete keys cause a rubout of the last character when input is through the string input functions.

4. Regardless of the source of the data, if the character is a line feed it is discarded, and if it is a carriage return it is changed to the value of the Small C newline character (**'\n'**). The effect is to represent the end of a line with a single newline character.

5. Regardless of the source of the data, if the character is a <control-Z> the end-of-file status is set and the character is changed to **EOF,** as defined in **STDIO.H.**

On output, the following occurs:

1. If the character has the value **EOF,** it is changed to the DOS end-of-file character—<control-Z>.

2. If the character is the newline character (**'\n'**), it is written as the sequence: carriage return, line feed.

Notice how the end of a line is handled. While it may be represented externally by two characters, internally it is represented by just one character with the value of the character constant **'\n'** (see Chapter 3). This two-to-one mapping is a standard feature of ASCII I/O transfers in the C language. Although the value of **'\n'** is defined by **NEWLINE** in the header file **STDIO.H,** we should always use the escape sequence rather than **NEWLINE** symbol or its value when testing for the end of a line. Not all C compilers assign the same value to **'\n'** and none of them define **NEWLINE** in their **STDIO.H** files.

The backspace and <control-X> characters have special meanings to the string input functions **fgets()** and **gets().** If input is from the keyboard, a backspace (recall that DEL becomes a backspace) causes these functions to *rubout* the last character received. In so doing they echo a backspace, space, backspace sequence to the screen to erase the previous character and reposition the cursor over it. Furthermore, a <control-X> from the keyboard causes them to *wipeout* the entire line in a similar manner. This allows the operator to restart his input from scratch. These functions are also sensitive to **EOF** (externally <control-Z>) as the end-of-file indicator. Upon receipt of this value, they cease input, terminate the input string with a null byte, and return to the user.

Remember that only the **get** and **put** functions cook the data. The **read** and **write** functions simply transfer it without change. In addition, there is a special function called **poll()** which polls the keyboard without waiting for a key to be struck.

This scheme of dividing the data transfer functions into ASCII and binary groups is unique to Small C. Full C compilers do this by providing open modes that indicate whether or not cooking is to be done. When porting Small C programs to Full C compilers, the new open modes must specify the same actions as the Small C functions.

We now look at the I/O functions individually.

```
fopen (name, mode) char *name, *mode;
```

This function attempts to open the file named by the character string pointed to by **name. Name** is an ordinary MS-DOS file specification, possibly including drive, path, filename, and extension. Only the filename is required. The drive and path will assume the defaults as established by MS-DOS. Obviously, the wildcard characters (**?** and *****) are not allowed.

Mode points to a string indicating the use for which the file is being opened. The following values are accepted:

"r"	read
"w"	write
"a"	append

Read mode opens an existing file for input.

Write mode creates a new file. If a file with the same name already exists, it is replaced by the new file. Since the new file starts out empty, writing begins with the first byte.

Append mode allows writing which begins at the end of an existing file or the beginning of a new one. If a file with the same name already exists, it is opened and positioned after the last byte. If no file with the specified name is found, one is created.

In addition, there are variations on these three modes which permit both reading and writing (updating). They are:

"r+"	update read
"w+"	update write
"a+"	update append

In terms of their effects at open time, these modes are the same as their non-update counterparts; however, these modes support both input and output operations. We can freely mix them in any order that we find useful. Each file has a

current position from which the next read or write operation begins. After each operation, the current position is set to the byte following the last one transferred. Therefore, successive I/O operations reference successive byte strings in the file. This sequential behavior can be altered by calls to functions which directly alter the current position to provide a random accessing capabilities.

If the attempt to open a file is successful, **fopen()** returns an **fd** value for the open file; otherwise, it returns **NULL.** The **fd** should be kept for use with subsequent I/O function calls. A typical statement that opens a file might look like:

```
if(!(data_fd = fopen("data.fil", "r")))
     error("can't open: data.fil");
```

where **error()** is an error handling function in the program. Only the standard files may be used without first calling **fopen().**

```
fclose (fd) int fd;
```

This function closes the specified file. Since Small C buffers disk file data, when a disk file (opened for writing or updating) is closed, its buffer is flushed to the disk if it contains pending data. This function returns **NULL** if the file closes successfully, and **ERR** if an error occurs.

When a program exits, either by reaching the end of **main()** or by calling **exit()** or **abort(),** all open files are closed automatically. However, when a program dies a hard death (loss of system power or rebooting the system), buffered data may be lost.

```
freopen (name, mode, fd) char *name, *mode; int fd;
```

This function closes the previously opened file indicated by **fd** and opens a new one whose **name** is in the character string at **name. Mode** specifies the new open mode just as for **fopen().** On success, this function returns the original value of **fd.** On failure, however, it returns **NULL.**

Note that since the **fd** for the standard input file is zero, there is no way of distinguishing success from failure in that case. It is best to simply leave **stdin** open even if it will not be used.

```
getc (fd) int fd;
fgetc (fd) int fd;
```

This function has two names. It returns the next character from the file indicated by **fd.** If no more characters remain in the file, or an error condition occurs, it returns **EOF.** The end of the file is detected by an occurrence of the end-of-file character (<control-Z>) or the physical end of the file.

```
getchar ()
```

This function is equivalent to the call **fgetc(stdin).** It presumes to use the standard input file.

```
fgets (str, sz, fd) char *str; int sz, fd;
```

This function reads up to **sz-1** characters into memory from the file indicated by **fd.** The target buffer in memory is indicated by **str.** Input terminates after transferring a newline character. A null character is appended behind the newline. If no newline is found after transferring **sz-1** characters, input also terminates and a null character is appended after the last character transferred. **Fgets()** returns **str** if the operation is successful. If the end of the file is reached while attempting to obtain another character, or if an error occurs, it returns **NULL.**

```
ungetc (c, fd) char c; int fd;
```

This function logically (not physically) pushes the character **c** back into the input file indicated by **fd.** The next read from the file will retrieve the ungotten character first. Only one character at a time (per file) may be kept waiting in this way. This function returns the character itself on success, or **EOF** if a previously pushed character is being held or if **c** has the value **EOF.** We cannot push **EOF** back into a file. Performing a seek or rewind operation on a file causes the ungotten character to be forgotten.

```
fread (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;
```

This function reads, from the file indicated by **fd, cnt** items of data of length **sz.** The target buffer is indicated by **ptr.** A binary transfer is performed and only the physical end of the file is recognized. A count of the actual number of items

read is returned to the caller. This could be less than **cnt** if the end of the file is reached. This is the usual way to tell when the end of the file is reached. However, we may call **feof()** to determine when the data has been exhausted, and **ferror()** to detect errors.

```
read (fd, ptr, cnt) int fd, cnt; char *ptr;
```

This function reads, from the file indicated by **fd, cnt** bytes of data into memory at the address indicated by **ptr.** This function performs a binary transfer and recognizes only the physical end of the file. A count of the actual number of bytes read is returned to the caller. This could be less than **cnt** if the end of the file was encountered. This is the usual way of telling when the end of the file is reached. However, we may call **feof()** to determine for certain when the data is exhausted, and **ferror()** to detect errors.

```
gets (str) char *str;
```

This function reads characters from **stdin** into memory starting at the address indicated by **str.** Input is terminated at the end of a line, but the newline character is not transferred. A null character terminates the input string. **Gets()** returns **str** for success; otherwise, it returns **NULL** for end-of-file or an error. Since this function has no way of knowing the size of the destination buffer, it is possible that an input line might overrun the allotted space. We must check the size of the input string to verify that it was not too large.

```
putc (c, fd) char c; int fd;
fputc (c, fd) char c; int fd;
```

These identical functions write the character **c** to the file indicated by **fd.** They return the character itself on success; otherwise, **EOF.**

```
putchar (c) char c;
```

This function is equivalent to the call **fputc (c, stdout).** It presumes to use the standard output file.

```
fputs (str, fd) char *str; int fd;
```

This function writes a string of characters, beginning at the address indicated by **str** to the file indicated by **fd.** Successive characters are written until a null byte is found. The null byte is not written and a newline character is not appended to the output. It is the programmer's responsibility to see that the necessary newline characters are in the string itself.

```
puts (str) char *str;
```

This function is almost like the call **fputs(str, stdout).** It presumes to write to the standard output file. Unlike **fputs(),** however, it appends a newline to the output.

```
fwrite (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;
```

This function writes to the file indicated by **fd, cnt** items of **sz** bytes from the memory buffer indicated by **ptr.** It returns a count of the number of items written. An error condition may cause the number of items written to be less than **cnt. Ferror()** should be called to verify error conditions, however. This function performs a *binary* transfer.

```
write (fd, ptr, cnt) int fd, cnt; char *ptr;
```

This function writes, to the file indicated by **fd, cnt** bytes from the memory buffer indicated by **ptr.** It returns a count of the number of bytes written. An error condition may cause the number of items written to be less than **cnt. Ferror()** should be called to verify error conditions, however. This function performs a *binary* transfer.

```
printf (str, arg1, arg2, ...) char *str;
```

This function writes to the standard output file, a formatted string which is the string at **str** with the ASCII equivalents of **arg1, arg2, ...** inserted at specified points.

The function returns a count of the total number of characters written. The string at **str** is called the *control string.* The control string is required, but the other arguments are optional. The control string contains ordinary characters and

groups of characters called *conversion specifications*. Each conversion specification tells **printf()** how to convert its corresponding argument into an ASCII string for output. The converted argument replaces its conversion specification in the output. The character **%** signals the start of a conversion specification and one of the letters **b, c, d, o, s, u,** or **x** terminates it.

Between the start and end of a conversion specification, the following optional fields may be found (in the order listed, without intervening blanks):

1. a minus sign **(-),**
2. a decimal integer constant **(nnn),** and/or
3. a decimal fraction **(.mmm)**

These subfields are all optional. In fact, one frequently sees conversion specifications without them.

The minus sign (-) indicates that the string, produced by applying a specified conversion to its argument, is to be left adjusted in its output field.

The decimal integer (**nnn**) indicates the minimum width of the output field (in characters). If more space is needed it will be used, but at least the number of positions indicated will be generated.

The decimal fraction (**.mmm**) is used where the argument is itself a character string (more correctly, the address of a character string). In this case, the fraction indicates the maximum number of characters to take from the string. If there is no fraction in the specification, then all of the string is used.

The terminating letter indicates the type of conversion to be applied to the argument. It may be one of the following:

b The argument should be considered an unsigned integer and converted to *binary* format for output. No leading zeroes are generated. This specification is unique to Small C.

c The argument should be output as a *character* without conversion. The high-order byte is to be ignored.

d The argument should be considered a signed integer and converted to a (possibly signed) *decimal* digit string for output. No leading zeroes are generat-

ed. The left-most character is reserved for the sign—blank if positive, hyphen if negative.

o The argument should be considered an unsigned integer and converted to *octal* for output. No leading zeroes are generated.

s The argument is the address of a *string* which should be output according to the justification, minimum width, and maximum size specifications indicated.

u The argument should be considered an unsigned integer and converted to *unsigned* decimal for output. No leading zeroes are generated.

x The argument should be considered an unsigned integer and converted to *hexadecimal* for output. No leading zeroes are generated.

If a **%** is followed by anything other than a valid specification, it is ignored and the following character is written without change. Thus, **%%** writes **%.**

Printf() scans the control string from left to right, sending everything to **stdout** until it finds a **%.** It then evaluates the conversion specification and applies it to the first argument (following the control string). The resulting string is written to **stdout.** It then resumes writing from the control string until it finds the next conversion specification which it applies to the second argument. This continues until the control string is exhausted. The result is a formatted output message consisting of both literal and variable data. See Table 12-2 for examples.

Control String	Arguments	Output
"oct %o, dec %d"	127, 127	oct 177, dec 127
"%u=%x"	-1, -1	65535=FFFF
"%d%% interest"	10	10% interest
"(%6d)"	55	(55)
"(%-6d)"	123	(123)
"The letter is %c."	'A'	The letter is A.
"Call me %s."	"Fred"	Call me Fred.
"Call me %.3s."	"Fred"	Call me Fre.
"Call me %4.3s."	"Fred"	Call me Fre.

Table 12-2. Printf Examples

```
fprintf (fd, str, arg1, arg2, ...) int fd; char *str;
```

This function works exactly like **printf()** except that output goes to the file indicated by **fd** instead of **stdout.**

```
scanf (str, arg1, arg2, ...) char *str;
```

This function reads a series of fields from the standard input file, converts them to internal format according to conversion specifications contained in the control string **str,** and stores them at the locations indicated by the arguments **arg1, arg2,....**

It returns a count of the number of fields read. A field in the input stream is a contiguous string of graphic characters. It ends with the next white space (blank, tab, or newline) or, if its conversion specification indicates a maximum field width (below), it ends when the field width is exhausted. A field normally begins with the first graphic character after the previous field; that is, leading white space is skipped. Since the newline character is skipped while searching for the next field, **scanf()** reads as many input lines as necessary to satisfy the conversion specifications in the control string. Each of the arguments following the control string must yield an address value.

The control string contains conversion specifications and white space (which is ignored). Each conversion specification informs **scanf()** how to convert the corresponding field into internal format, and each argument following **str** gives the address where the corresponding converted field is to be stored. The character **%** signals the start of a conversion specification and one of the letters **b, c, d, o, s, u,** or **x** ends it.

Between these may be found, with no intervening blanks, an asterisk and/or a decimal integer constant. These subfields are both optional. In fact, conversion specifications are frequently written without them.

The asterisk indicates that the corresponding field in the input stream is to be skipped. Skip specifications do not have corresponding arguments.

The numeric field indicates the maximum field width in characters. If present, it causes the field to be terminated when the indicated number of characters has been scanned, even if no white space is found. However, if white space is

found before the field width is exhausted, the field is terminated at that point.

The terminating letter indicates the type of conversion to be applied to the field. It may be one of the following:

b The field should be considered a *binary* integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored. This specification is unique to Small C.

c The field should be accepted as a single *character* without conversion. This specification inhibits the normal skip over white space; blanks are transferred just like other characters. The argument for such a field should be a character address.

d The input field should be considered a (possibly signed) *decimal* integer and converted into an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored.

o The field should be considered an unsigned *octal* integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored.

s The field should be considered a character *string* and stored with a null terminator at the character address indicated by its argument. There must be enough space at that address to hold the string and its terminator. Remember, a maximum field width can be specified to prevent overflow. The specification **%1s** will read one character. It differs from **%c** in that it skips white space, whereas the latter reads the next character, whatever it is.

u The field should be considered an *unsigned* decimal integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored. This specification is unique to Small C.

x The field should be considered an unsigned *hexadecimal* number and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes or a leading **0x** or **0X** will be ignored.

Scanf() scans the control string from left to right, processing input fields until the control string is exhausted or a field is found which does not match its

conversion specification. If the value returned by **scanf()** is less than the number of conversion specifications, an error has occurred or the end of the input file has been reached. **EOF** is returned if no fields are processed because the end of the file has been reached.

If the statement,

```
scanf("%s   %c   %c   %*s   %d   %3d   %d",
    str, &c1, &c2,       &i1,  &i2, &i3);
```

is executed when the input stream contains:

```
"abc defg    -12   345678 9"
```

the values stored would be:

```
"abc\0" at str
  ' ' in c1
  'd' in c2
  -12 in i1
  345 in i2
  678 in i3
```

Future input from the file would begin with the space following **345678.**

```
fscanf (fd, str, arg1, arg2, ...) int fd; char *str;
```

This function works like **scanf()** except that the input is taken from the file indicated by **fd,** instead of **stdin.**

```
rewind (fd) int fd;
```

This function positions the file indicated by **fd** to its beginning. It is equivalent to a seek to the first byte of the file. It returns **NULL** on success; otherwise, **EOF.**

```
bseek (fd, offset, from) int fd, offset[], from;
```

This Small C function positions the MS-DOS file pointer for **fd** to the byte indicated by a double length integer offset. **Offset** must be given as the address of an integer array of two elements such that **offset[0]** is the low-order half and **offset[1]** is the high-order half. **From** determines the base location from which the offset is applied. It may have one of the values:

0 beginning of file
1 current byte in file
2 end of file

The last case should be used with a minus offset.

Bseek() returns **NULL** on success; otherwise, **EOF.** It works like the UNIX **fseek()** except that the offset is an address instead of the actual value. This is necessary since Small C does not support long variables.

```
btell (fd, offset) int fd, offset[];
```

This Small C function returns (tells) the offset to the current byte in the file indicated by **fd.** The current byte is the next one that will be read from or written to the file. No account of **ungetc()** calls is taken. **Offset** is a double length integer and must be specified as the address of an integer array of two elements such that **offset[0]** is the low-order half and **offset[1]** is the high-order half. The array at offset receives the offset value of the current byte in the file. **Btell()** returns **NULL** on success; otherwise, **ERR.**

```
cseek (fd, offset, from) int fd, offset, from;
```

This Small C function positions the file indicated by **fd** to the beginning of the 128-byte block which is **offset** positions from the first block, current block, or end of the file depending on whether **from** is 0, 1, or 2, respectively. Subsequent reads and writes proceed from that point. **cseek()** returns **NULL** for success; otherwise, **EOF.**

```
cseekc (fd, offset) int fd, offset;
```

This Small C function positions **fd** on the character indicated by **offset** within the current 128-byte block. The current byte must be on a block boundary when this function is called. Used in combination with **cseek()** this function allows us to seek to any given byte in a file by tracking file positions in two parts—the block offset and the byte offset within the block. **cseekc()** returns **NULL** on success; otherwise, **EOF.**

```
ctell (fd) int fd;
```

This Small C function returns the position of the current block of the file indicated by **fd.** The returned value is the offset of the current 128-byte block with respect to the beginning of the file. If **fd** is not assigned to a disk file, **ERR** is returned.

```
ctellc (fd) int fd;
```

This Small C function returns the offset (0-127) to the current byte in the current block for **fd.** The current byte is the next one that will be read from or written to the file. No account of **ungetc()** calls is taken.

```
rename (old, new) char *old, *new;
```

This Small C function changes the name of the file specified by **old** to the name indicated by **new.** It returns **NULL** on success; otherwise, **ERR.**

```
delete (name) char *name;
unlink (name) char *name;
```

This function deletes the file indicated by the character string at **name.** It returns **NULL** on success; otherwise, **ERR.**

```
feof (fd) int fd;
```

This function returns *true* if the file designated by **fd** has reached its end. Otherwise, it returns *false*.

```
ferror (fd) int fd;
```

This function returns *true* if the file designated by **fd** has encountered an error condition since it was last opened. Otherwise, it returns *false*.

```
clearerr (fd) int fd;
```

This function clears the error status for the file indicated by **fd.**

```
iscons (fd) int fd;
```

This Small C function returns *true* if **fd** is assigned to the console; otherwise, *false*.

```
isatty (fd) int fd;
```

This function returns *true* if **fd** is assigned to a device rather than a disk file; otherwise, *false*.

```
auxbuf (fd, size) int fd, size;
```

This Small C function allocates an auxiliary buffer of **size** bytes for use with **fd.** It returns zero for success, and **ERR** if it fails. **fd** may or may not be open. **Size** must be greater than zero and less than the amount of free memory. If **fd** is a device, the buffer is allocated, but ignored. This implementation of Small C uses *file handle*-type MS-DOS calls; it reads and writes in chunks the size of the auxiliary buffer.

Extra buffering is useful in reducing disk head movement and drive switching during sequential operations. Once an auxiliary buffer is allocated, it remains for the duration of program execution, even if **fd** is closed. Calling this function a second time for **fd** is allowed; however, the original buffer continues to occupy space in memory.

Alternating read and write operations and performing seeks with auxiliary buffering is also allowed. **Ungetc()** will operate normally. Ordinarily, it is unnecessary and wasteful to allocate auxiliary buffers to both input and output files.

Format Conversion Functions

```
atoi (str) char *str;
```

This function converts the decimal number in the string at **str** to an integer, and returns its value. Leading white space is skipped and an optional sign (+, -) may precede the left-most digit. The first non-numeric character terminates the conversion.

```
atoib (str, base) char *str; int base;
```

This Small C function converts the unsigned integer of base **base** in the string at **str** to an integer, and returns its value. Leading white space is skipped. The first non-numeric character terminates the conversion.

```
itoa (nbr, str) int nbr; char *str;
```

This function converts the number **nbr** to its decimal string representation at **str.** The result is left justified at **str** with a leading minus sign if **nbr** is negative. A null character terminates the string, which must be large enough to hold the result.

```
itoab (nbr, str, base) int nbr; char *str; int base;
```

This Small C function converts the unsigned integer **nbr** to its string representation at **str** in base **base.** The result is left justified at **str.** A null character terminates the string, which must be large enough to hold the result.

```
dtoi (str, nbr) char *str; int *nbr;
```

This Small C function converts the (possibly signed) decimal number in the string at **str** to an integer at **nbr** and returns the length of the numeric field found. The conversion stops when the end of the string or a non-decimal character is reached. **Dtoi()** will use a leading sign and at most, five digits. **Dtoi()** returns **ERR** if the absolute value of the number exceeds 32767.

```
otoi (str, nbr) char *str; int *nbr;
```

This Small C function converts the octal number in the string at **str** to an integer at **nbr** and returns the length of the octal field. The conversion stops

when the end of the string or a non-octal character is reached. **Otoi()** will use six digits at most. A number larger than 177777 will cause **otoi()** to return **ERR.**

```
utoi (str, nbr) char *str; int *nbr;
```

This Small C function converts the unsigned decimal number in the string at **str** to an integer at **nbr,** and returns the length of the numeric field. The conversion stops when the end of the string or a non-decimal character is reached. **Utoi()** will use five digits at most. A number larger than 65535 will cause **utoi()** to return **ERR.**

```
xtoi (str, nbr) char *str; int *nbr;
```

This Small C function converts the hexadecimal number in the string at **str** to an integer at **nbr** and returns the length of the hexadecimal field. The conversion stops when the end of the string or a non-hexadecimal character is reached. **Xtoi()** will use four digits at most. If more hex digits are present, it returns **ERR.**

```
itod (nbr, str, sz) int nbr, sz; char *str;
```

This Small C function converts **nbr** to a (possibly) signed character string at **str.** The result is right-justified and blank filled in **str.** The sign and possibly high-order digits are truncated if the destination string is too small. It returns **str. Sz** indicates the length of the string. If **sz** is greater than zero, a null byte is placed at **str[sz-1].** If **sz** is zero, a search for the first null byte following **str** locates the end of the string. If **sz** is less than zero, all **sz** characters of **str** are used including the last one.

```
itoo (nbr, str, sz) int nbr, sz; char *str;
```

This Small C function converts **nbr** to an octal character string at **str.** The result is right justified and blank filled in the destination string. High-order digits are truncated if the destination string is too small. It returns **str. Sz** indicates the length of the string. If **sz** is greater than zero, a null byte is placed at **str[sz-1].** If **sz** is zero, a search for the first null byte following **str** locates the end of the string. If **sz** is less than zero, all **sz** characters of **str** are used.

```
itou (nbr, str, sz) int nbr, sz; char *str;
```

This Small C function converts **nbr** to an unsigned decimal character string at **str.** It works like **itod()** except that the high-order bit of **nbr** is taken as a magnitude bit.

```
itox (nbr, str, sz) int nbr, sz; char *str;
```

This Small C function converts **nbr** to a hexadecimal character string at **str.** The result is right-justified and blank filled in the destination string. High-order digits are truncated if the destination string is too small. It returns **str. Sz** indicates the length of the string. If **sz** is greater than zero, a null byte is placed at **str[sz-1].** If **sz** is zero, a search for the first null byte following **str** locates the end of the string. If **sz** is less than zero, all **sz** characters of **str** are used, including the last one.

String Manipulation Functions

```
left (str) char *str;
```

This Small C function left-adjusts the character string at **str.** Starting with the first non-blank character and proceeding through the null terminator, the string is moved to the address indicated by **str.** This function can be used to left-adjust the output of the **ito?()** functions above. For example,

```
left(itod(i, str, 6));
```

will convert **i** to a decimal string, which is left-adjusted in **str.**

```
strcat (dest, sour) char *dest, *sour;
```

This function appends the string at **sour** to the end of the string at **dest.** The null character at the end of **dest** is replaced by the leading character of **sour.** A null character terminates the new **dest** string. The space reserved for **dest** must be large enough to hold the result. This function returns **dest.**

```
strncat (dest, sour, n) char *dest, *sour; int n;
```

This function works like **strcat()** except that a maximum of **n** characters from the source string will be transferred to the destination string.

```
strcmp (str1, str2) char *str1, *str2;
```

This function returns an integer less than, equal to, or greater than zero depending on whether the string at **str1** is less than, equal to, or greater than the string at **str2.** The strings are compared left to right, character by character, until a difference is found or they end simultaneously. Comparison is based on the numeric values of the characters. **Str1** is considered less than **str2** if **str1** is equal to but shorter than **str2,** and vice versa.

```
strncmp (str1, str2, n) char *str1, *str2; int n;
```

This function works like **strcmp()** except that a maximum of **n** characters are compared.

```
strcpy (dest, sour) char *dest, *sour;
```

This function copies the string at **sour** to **dest. Dest** is returned. The space at **dest** must be large enough to hold the string at **sour.** A null character follows the last character placed in the destination string.

```
strncpy (dest, sour, n) char *dest, *sour; int n;
```

This function works like **strcpy()** except that **n** characters are placed in the destination string regardless of the length of the source string. If the source string is too short, null padding occurs. If it is too long, it is truncated in **dest.** A null character follows the last character placed in the destination string.

```
strlen (str) char *str;
```

This function returns a count of the number of characters in the string at **str.** It does not count the null character that terminates the string. Since the length of a string can be found only by scanning it from beginning to end, this function can be time-consuming. To minimize this overhead, the current Small C library implements this function in assembly language with a repeat prefix attached to

the 8086 *string compare* instruction for the fastest possible speed.

```
strchr (str, c) char *str, c;
```

This function returns a pointer to the first occurrence of the character **c** in the string at **str.** It returns **NULL** if the character is not found. Searching ends at the null terminator.

```
strrchr (str, c) char *str, c;
```

This function works like **strchr()** except that the right-most occurrence of the character is sought.

```
reverse (str) char *str;
```

This function reverses the order of the characters in the null terminated string at **str.**

```
pad (str, ch, n) char *str, ch; int n;
```

This Small C function fills the string at **str** with **n** occurrences of the character ch.

Character Classification Functions

The following functions determine whether or not a character belongs to a designated class of characters. They return *true* if it does, and *false* if not. Since these functions are identical except for the condition for which they test, they are simply listed below with their conditions:

Except for **isascii(),** only characters in the ASCII set (0–127) should be tested. Characters greater than 127 decimal yield unpredictable answers.

`isalnum (c) char c;`	alphanumeric ('A'–'Z', 'a'-'z', '0'–'9')
`isalpha (c) char c;`	alphabetic ('A'–'Z', 'a'–'z')
`isascii (c) char c;`	ASCII (0-127)
`iscntrl (c) char c;`	control character (0-31, 127)
`isdigit (c) char c;`	digit ('0'–'9')
`isgraph (c) char c;`	graphic (33–126)

```
islower (c) char c;
isprint (c) char c;
ispunct (c) char c;
isspace (c) char c;
isupper (c) char c;
isxdigit (c) char c;
```

lowercase letter ('a'–'z')
printable (32–126)
punctuation (not cntrl, alnum, space)
white space (SP, HT, VT, CR, LF, or FF)
uppercase letter ('A'–'Z')
hexadecimal digit ('0'–'9', 'A'–'F', 'a'-'f')

Character Translation Functions

```
toascii (c) char c;
```

This function returns the ASCII equivalent of **c.** Since MS-DOS systems use the ASCII character set, it merely returns **c** unchanged. This function makes it possible to use the properties of the ASCII code set without introducing implementation dependencies into programs.

```
tolower (c) char c;
```

This function returns the lowercase equivalent of **c** if **c** is an uppercase letter. Otherwise, it returns **c** unchanged.

```
toupper (c) char c;
```

This function returns the uppercase equivalent of **c** if **c** is a lowercase letter. Otherwise, it returns **c** unchanged.

Lexicographical Comparison Functions

```
lexcmp (str1, str2) char *str1, *str2;
```

This Small C function works like **strcmp()** except that a lexicographical comparison is used. For meaningful results, only characters in the ASCII character set (codes 0–127) should appear in the strings. Alphabetic characters are compared without case sensitivity; i.e., uppercase and lowercase letters are equivalent. Overall, the sequence is: (1) control characters, (2) special characters (in ASCII order), (3) numerics, (4) alphabetics, and (5) the delete character (DEL).

```
lexorder (c1,c2) char c1, c2;
```

This Small C function returns an integer less than, equal to, or greater than zero depending on whether **c1** is less than, equal to, or greater than **c2** lexicographically. For meaningful results, only characters in the ASCII character set (codes 0–127) should be passed. Alphabetic characters are compared without case sensitivity; i.e., uppercase and lowercase letters are equivalent. Overall, the sequence is: (1) control characters, (2) special characters (in ASCII order), (3) numerics, (4) alphabetics, and (5) the delete character (DEL).

Mathematical Functions

```
abs (nbr) int nbr;
```

This function returns the absolute value of **nbr.**

```
sign (nbr) int nbr;
```

This function returns **-1, 0,** or **+1** depending on whether **nbr** is less than, equal to, or greater than zero, respectively.

Program Control Functions

```
avail (abort) int abort;
```

This Small C function returns the number of bytes of free memory which exists between the memory heap and the stack. It also checks to see if the stack overlaps the heap; if so, and if **abort** is not zero, the program is aborted with an exit code of 1. However, if **abort** is zero, **avail()** returns zero to the caller. This function makes it possible to make full use of the data segment. However, care should be taken to leave enough space for the stack to grow.

```
calloc (nbr, sz) int nbr, sz;
```

This function allocates **nbr*sz** bytes of zeroed memory. It returns the address of the allocated memory block. If insufficient memory exists, the request fails and zero is returned. If the stack and the heap are found to overlap, then the program aborts with an exit code of 1.

```
malloc (nbr) int nbr;
```

This function allocates **nbr** bytes of uninitialized memory. It returns the address of the allocated memory block. If insufficient memory is available, the request fails and zero is returned. If the stack and the heap are found to overlap, then the program aborts with an exit code of 1.

```
free (addr) char *addr;
cfree (addr) char *addr;
```

This function releases a block of previously allocated heap memory beginning at **addr.** It returns **addr** on success; otherwise **NULL.**

Small C uses a simplified memory allocation scheme. It allocates memory in a heap that starts after the last global item in the data segment, and expands in the direction of the stack. Furthermore, it does not keep track of memory which has already been allocated; it only maintains a single pointer to the end of the heap. Because of this, blocks of allocated memory must be freed in the reverse order from which they were allocated.

Since the open functions allocate buffer space on the heap and the close function does not free it, freeing memory which was allocated before opening a file must be avoided.

```
getarg (nbr, str, sz, argc, argv)
  char *str; int nbr, sz, argc, *argv;
```

This Small C function locates the command-line argument indicated by **nbr,** moves it (null terminated) into the string at **str** (maximum size **sz**), and returns the length of the field obtained. **Argc** and **argv** must be the same values provided to **main()** when the program is started. If **nbr** is one, the first argument (following the program name) is requested; if two, the second argument is requested, and so on. If there is no argument for **nbr, getarg()** puts a null byte at **str** and returns **EOF.**

```
poll (pause) int pause;
```

This Small C function polls the keyboard for operator input. If no input is waiting, zero is returned. If a character is waiting, the value of **pause** determines

what happens. If **pause** is zero, the character is returned immediately. If **pause** is not zero and the character is a <control-S>, there is a pause in program execution; when the next character is entered, zero is returned. If the character is a <control-C>, program execution is terminated with an exit code of 2. All other characters are returned to the caller immediately. **Poll()** calls **_getkey()** and translates auxiliary keystrokes accordingly (see Table 12-3).

```
abort (errcode) int errcode;
exit (errcode) int errcode;
```

This function closes all open files and returns to MS-DOS. The value of **errcode** determines whether or not a normal exit is to be taken. If it is not zero, then the message:

```
Exit Code: n
```

(where **n** is the decimal value of **errcode**) is displayed on the screen before control returns to MS-DOS. In either case, **errcode** is returned to MS-DOS for checking in batch files.

Primitive Functions

The following functions are not normally needed in programs, but they exist in the standard Small C library and are documented here in case they may be of use. Since they are in the module **CSYSLIB,** which is linked with every Small C program, there is no incremental memory cost associated with their use.

```
_hitkey()
```

This Small C function calls the Basic I/O System (BIOS) to test the keyboard for a pending keystroke. It returns *true* or *false* according to whether or not a key has been hit.

```
_getkey()
```

This Small C function calls the BIOS to return the next byte from the keyboard. It returns immediately if a keystroke is pending; otherwise, it waits for

one. By testing the keyboard with **_hitkey()** before calling this function, we can avoid blocking the program until a key is pressed. This is how **poll()** obtains its input. All input from the keyboard, regardless of the input function, is obtained through this function.

Most keystrokes have obvious ASCII values. However, there are numerous *auxiliary* codes that can be generated through the use of special keys and various key combinations. The BIOS assigns values to auxiliary keystrokes which overlap the ASCII codes. To distinguish between these, **_getkey()** adds an offset of 113 (decimal) to the BIOS's auxiliary codes, placing them in the range 128–245. The only exception is the traditional (with ASCII terminals) key combination <control-@> which usually generates a value of zero. The BIOS's auxiliary code for this (and the unshifted <ctrl-2>) is 3, which **_getkey()** translates directly to zero. Table 12-3 lists the special keystrokes and the values transmitted through the standard Small C input functions.

Key Combinations	Values	Key Combinations	Values
ctrl-2 (or ctrl-@)	0	Insert	195
shift-tab	128	Delete	196
alt-QWERTYUIOP	129-138	shift-F1/shift-F10	197-206
alt-ASDFGHJKL	143-151	ctrl-F1/ctrl-F10	207-216
alt-ZXCVBNM	157-163	alt-F1/alt-F10	217-226
F1/F10	172-181	ctrl-PrtSc	227
Home	184	ctrl-left arrow	228
up arrow	185	ctrl-right arrow	229
PgUp	186	ctrl-End	230
left arrow	188	ctrl-Home	232
right arrow	190	alt-1/alt-0	233-242
End	192	alt-hyphen	243
down arrow	193	alt-equal	244
PgDn	194	ctrl-PgUp	245

Table 12-3. Small C Codes for Auxiliary Keystrokes

```
_bdos2(ax, bx, cx, dx) int ax, bx, cx, dx;
```

This Small C function provides a general means of making interrupt-21 calls to the Basic Disk Operating System (BDOS). The digit **2** in its name signifies that **_bdos2()** is designed for use with version 2.0 MS-DOS services. Except for the keyboard servicing functions above, all other interactions with a Small C program's environment pass through this function.

This function takes four 16-bit arguments—the values to place in the AX, BX, CX, and DX registers before issuing the interrupt. On return, if there was no error, zero is returned. If something went wrong, however, the returned value contains the standard MS-DOS error code.

Since MS-DOS draws attention to the error code by setting the carry flag (although apparently not consistently), this function also passes on that information by negating the error codes. If the carry flag is set, indicating an error, the error codes are subtracted from zero to form a negative value (two's complement of the original). So, if we want to trust MS-DOS's carry flag error indicator, we can just test for a value less than zero. On the other hand, a more dependable test for errors is any non-zero value. If we want to analyze the specific error, then we must negate the code again if it is negative.

Refer to other material for details on the BDOS calls and the standard error codes. Peter Norton [15] and Ray Duncan [16] are good sources for this information.

CHAPTER 13

Efficiency Considerations

Ideally, our programs should be both efficient and well-designed. To a point, both objectives can be realized by careful planning before committing our designs to code. Once a significant effort has gone into a program, however, it becomes easier to live with what we have than to start over again when we discover a major flaw or a better algorithm. As a result, we make do with something that is less than it could have been. So the first rule for writing well-designed, efficient programs is to *think first.*

At various points in the planning phase, we must decide between good programming style and efficiency. We often have to make trade-off decisions. And even when the emphasis is on efficiency, we often have to choose between program size and speed.

As a rule, good programming style is more important than efficiency, because programming time is expensive and the more clearly a program is written the more quickly it can be documented and revised. However, there are times when it is more important for programs to work with limited memory and/or to satisfy fixed performance criteria.

This chapter does not deal with programming style; it merely provides information about the efficiency of certain C constructs as they are implemented by the Small C compiler, so that trade-off decisions can be made intelligently. The advice given below is not necessarily appropriate in all circumstances. Since, by default, Small C optimizes its output, the following comments relate to the optimized output, not the preliminary code produced by the **-NO** command-line switch (see Chapter 16).

Because of complexities (like instruction pipelining) in the design of the 80x86 family of processors, and the different instruction timings of the different

processors in the family, it is impossible to give exact information on the relative speeds of alternative constructs. So the following information, as it pertains to speed, should be taken as approximate.

Appendix B describes the 80x86 CPUs. It may help to review that material before proceeding with the following discussions, since they refer to the 80x86 registers.

Integer Alignment

Systems with an 8-bit data bus always require two memory accesses to fetch or store an integer or pointer. Systems with wider buses, however, can reference a word with just one memory access. With a 16-bit bus, one access fetches or stores both bytes if the word has an even address (i.e., is on a word boundary); but if the word has an odd address (i.e., is on a byte boundary), two memory accesses will be performed by the hardware since the word spans two aligned words in memory.

With a 32-bit bus, the situation is a bit more complicated. If the word is on a byte boundary, then it may either span two aligned double words in memory, or it may fall exactly in the middle of one. In the first case, two accesses are required; but, in the latter case one will suffice. So we can expect roughly half of our integers and pointers to be referenced efficiently, and half inefficiently. However, if an integer or pointer is word-aligned, then it must fall in the first or second half of an aligned double word. So, only one memory access is required.

We can take advantage of the way Small C allocates global storage to ensure that integers and pointers are word-aligned. Small C starts segments on word boundaries, and allocates storage in the order in which global definitions appear in the program. By writing all of our integer and pointer definitions before any character definitions we can ensure that all integers fall on word boundaries.

Unfortunately, Small C does not control the alignment of function arguments or local variables. So, although it may run counter to good programming practice, and if speed is important, we can declare integers and pointers globally rather than locally.

Chapter 28 contains a discussion on making the compiler align word-sized items in every case.

Integers and Characters

Compared to integers, fetching characters takes somewhat more time, and decidedly more instruction space, in memory. For instance, only one instruction:

```
MOV AX,_GI
```

(three bytes) is needed for a global integer, but two instructions:

```
MOV AL,_GC
CBW
```

(three bytes and one byte, respectively) are required for a global character. The second instruction converts the byte in AL to a word in AX by extending its sign bit through AH. If the character is declared to be unsigned, **CBW** is replaced with:

```
XOR AH,AH
```

(two bytes) which zeros AH.

Choosing character variables wherever they are sufficient does not necessarily reduce program size, and may, in fact, enlarge it somewhat. For instance, suppose we have the choice of defining a variable as a signed integer or a signed character, and suppose it is to be fetched five times in the program. Then we would save one byte of data space by declaring a character, but we would lose five bytes of code space. On the other hand, suppose that instead of a variable, we were considering an array of 1000 elements. In that case, declaring an integer array would cost 1000 bytes of data space (compared to a character array) while saving only five bytes of code space—not a wise choice. So, in situations where either integers or characters will do, it is usually better to declare integer variables and character arrays.

In terms of speed, the difference is not so clear. Suffice it to say that integers are generally accessed a little faster than characters because of the additional instruction that characters require. However, this tends to be compensated for by the need to transfer two bytes instead of one. On the other hand, the second byte may be free, as we saw above, depending on the data bus width and the alignment of the integers.

Since the storing of a character involves transferring just the low-order byte of AX to memory, there is no penalty associated with characters. On machines with 8-bit buses, or when the destination is aligned on a byte (not word) boundary, storing integers is slower than storing characters. However, they both require the same amount of instruction space—three bytes for global stores.

Globals and Locals

There is no appreciable difference in fetching or storing global and local objects. Fetching a global integer is done with:

```
MOV AX,_GI
```

and a local integer with:

```
MOV AX,-n[BP]
```

where **n** is an offset into the stack frame (see Chapter 8). In the first case, the instruction, together with the address of the global integer **_GI,** takes three bytes of space. Two bytes of this is the address which the processor uses to locate the operand. In the second case, the instruction together with the immediate value **n** also takes three bytes. In this case, the processor forms an effective address by adding **n** to **BP;** this takes very little time.

Global objects occupy the same amount of space as locals, and they always exist since they are static—even in the **EXE** file. On the other hand, locals are dynamically allocated at run time, so they require no space in the **EXE** file. Therefore, a program containing a large global array may waste disk space and take unnecessarily long to load. These disadvantages can be overcome two ways.

First, the array can be declared locally in **main().** If it requires initial values, then logic must be executed to initialize it; this is often a simple process, however, and the space it adds to the code segment may be small compared to what is saved in the data segment. The time to initialize such an array is negligible since the array is initialized only once. This method has the disadvantage that the array's address must be passed to every function that needs to refer to it. Another disadvantage is that each reference to the array, from functions other than **main(),** requires an additional memory access since the array is being referenced indirectly through a pointer—the address passed to the function.

A second approach is to declare a global pointer. Then, in **main(),** allocate memory for the array and assign its address to the pointer. From then on, the pointer can be used just as though it were an array. This method permits functions to refer to the array as a global object. Since the desired initial value for an array's elements is usually zero, the function **calloc()** can be used to allocate zeroed memory. The disadvantage of this method is that, as above, each reference to an element requires an additional memory access since the array is being referenced indirectly through the pointer; in this case a global pointer.

Constant Expressions

Since the compiler evaluates constant expressions at compile time, there is no penalty associated with writing constants as expressions. For instance, we may be working with a character array in which every four bytes constitutes a single entry of four one-byte fields. To make the program logic clearer, we could define symbols for the size of an entry **(#define SZ 4)** and for offsets to the individual fields in each entry (e.g., **#define FLD3 2**). If we have a pointer to some entry in the array (e.g., ***ap**), we could refer to the third field of the previous entry as:

```
*(ap + (FLD3 - SZ))
```

In this case, the expression only generates:

```
MOV BX,_AP        fetch address from pointer
MOV AL,-2[BX]     indirectly fetch character
CBW               sign extend to 16 bits
```

where the first instruction loads the contents of **ap** (an address) into BX, the second one moves into AL the byte located two bytes previous to that address, and the third one converts it to a word. Notice that the subexpression (**FLD3 - SZ**) appears as the offset **-2** in the second instruction. This is just as though we had written:

```
*(ap - 2)
```

Without the inner parentheses in this example, the compiler would have evaluated the expression as though it were written:

```
*((ap + FLD3) - SZ)
```

which is effectively the same, but not nearly as efficient. This generates:

```
MOV AX,_AP     fetch the address from pointer
ADD AX,2       add FLD3 to address
SUB AX,4       subtract SZ from address
MOV BX,AX      move address to BX
MOV AL,[BX]    indirectly fetch character
CBW            sign extend to 16 bits
```

Notice here that the constants appear in two places. Also, the optimizer is not smart enough to eliminate the fourth instruction by making the first three work directly with BX.

The point is that, because Small C evaluates constant (sub)expressions at compile time, there is no penalty associated with writing them in terms of their constituent parts. And whenever more than one constant appears in an expres-

sion, it is worth trying to group them together.

Remember from Chapter 9, that the **sizeof** operator yields a constant value, so it can be used in constant expressions.

Zero Tests

Whenever possible, you should write test expressions in **if, for, do,** and **while** statements so that they involve a comparison to the constant zero. In such cases, the compiler economizes on the code it generates. This is illustrated by the examples in Listing 19-23. The exact savings depends on which relational operator is involved, and whether a signed or unsigned comparison is performed.

Constant Zero Subscripts

The compiler adds a subscript to the address of an array or to the contents of a pointer in order to obtain the address of the specified element. However, if the compiler sees that the subscript has the constant value zero, it eliminates the addition altogether. It follows that, in terms of the code generated, writing **array[0]** is the same as writing ***array**—there is no penalty for using the subscript notation. This is true even if a constant expression is used for the subscript. If it evaluates to zero, the addition operation is skipped.

Switch Statements

Whenever possible, write a **switch** statement instead of a string of **if...else...** statements. The code generated by the compiler is much smaller since, for each condition, it generates only a pair of words (an address and a constant). At runtime, a library routine accepts the value of the expression and rapidly runs down the list of constants looking for a match. When it finds one, it branches to the corresponding address. With the **if...else...** approach, however, the expression must be evaluated and the result tested in each case. See Listing 19-22 for an example of the code generated by an **if...else...** sequence, and Listing 19-24 for a **switch** statement.

Pointer References

Generally, it is more efficient to use pointer references rather than subscripted references. The advantage is that pointer references do not have to involve the addition of a subscript, since the pointer itself can be adjusted like a subscript. For example, the reference:

```
*gip
```

where **gip** is a global integer pointer, generates:

```
MOV BX,_GIP   fetch pointer value
MOV AX,[BX]   indirectly fetch integer
```

On the other hand, the subscripted reference:

```
gia[gi],
```

where **gia** is a global integer array, and **gi** is a global integer, generates:

```
MOV AX,_GI                fetch subscript
MOV BX,OFFSET _GIA        fetch array address
SHL AX,1                  multiply subscript by 2
ADD BX,AX                 add subscript to address
MOV AX,[BX]               indirectly fetch integer
```

CHAPTER 14

Compatibility with Full C

The differences between Small C and full implementations of the language are important to programmers who anticipate using a Full C compiler in the future. No one wants to write programs that will be hard to convert. This *upward* compatibility is made possible by the fact that Small C is a subset compiler. For the same reason, however, *downward* conversions can be expected to be difficult or even out of the question.

This chapter presents these differences so that you will know how to write programs that easily convert to Full C and so you can estimate the difficulty of porting Full C programs to Small C.

The Big Differences

Of course, the major differences between Small C and Full C are the features of the C language that Small C does not support. These differences have no effect on upward portability of programs, but can make downward portability either difficult or unrealistic.

Small C's most significant limitations are on the data types it supports—integers, characters, pointers, and single-dimension arrays of integers or characters. Clearly, Full C programs which make use of unsupported data types are not candidates for conversion to Small C.

The next most significant limitations are its lack of support for structures and unions. A *structure* is a collection of objects of any type; other languages often call them *records*. A *union* is an object that has multiple declarations; that is, a single piece of memory that can contain any one of several types of data at different times. These limitations are less significant than the previously mentioned ones because they impose no hard limits on the language in terms of its

usefulness. The use of structures and unions is never essential, although it may seem so to programmers who depend on them. We can always declare a character array and place in it whatever kinds of data or collections of data we wish. The Small C symbol table is an example (see Chapter 20). There is no fundamental limitation to this technique, although it is less convenient than we might desire.

Full C programs which use structures can be ported down to Small C, but with some effort. First, the structures must be replaced with arrays of sufficient size. Then, if mixed character and integer data are to coexist in an array, functions (like **getint()** and **putint()** in the compiler) could be written to simplify accessing the array's contents. Finally, a routine search of the program should be performed to find and convert all of the references to the redefined structures. This last step is made easier by using the search facilities of a good text editor.

Undeclared Identifiers

During expression evaluation, the Small C compiler assumes that any undeclared name is a function, and automatically declares it as such. If the reference is followed by parentheses, a call is generated; otherwise, the function's address is generated by reference to the label bearing its name. If the same function is defined later in the program, the label for the function is generated. If, on the other hand, it is not defined, then Small C automatically declares the name as an external reference, to be resolved at link time. This arrangement makes it unnecessary to declare a function before referring to it.

Some Full C compilers assume this only if the name is written as a function call (with parentheses). Others gripe even in those cases, and require that we either avoid forward references or predeclare functions that are defined later in the program.

How we handle this difference is largely a matter of preference. We might wish to postpone predeclaring function names until we actually convert to Full C. Then, after the new compiler complains, insert the necessary function declarations at the front of the source file. That way we let the compiler hunt down the problem function names for us.

Function Names as Arguments

Small C accepts:

```
int arg
```

to declare a formal argument which points to a function, and:

```
arg (...)
```

to call the function. This is because Small C is not particular about what sort of expression precedes the parentheses that specify a function call. It simply evaluates the expression and uses the result as the offset in the code segment to the desired function.

Since function addresses are actually passed as pointers, a better syntax, which is compatible with Full C, is:

```
int (*arg)()
```

and,

```
(*arg)(...)
```

respectively. This syntax should be used to maintain compatibility with other compilers.

Indirect Function Calls

As we saw above, any expression followed by parentheses is taken by Small C as a function call, whereas Full C accepts only primary expressions based on a function name—like **(*func)** or **(*fa[x])**, Small C is not so particular. For instance, it will accept:

```
ia[x](...)
```

to call a function whose address is found in element **x** of an integer array **ia.**

This would not be accepted by a Full C compiler because **ia[x],** not being in parentheses, is not a primary expression and does not yield a function type. Small C will also accept this call rewritten as:

```
(*ia[x])()
```

This is as far as Small C can go since it does not know about pointer arrays. By writing the call this way, at conversion time it will only be necessary to change the declaration of the array when the program is ported to Full C.

Argument Passing

Small C differs from Full C compilers in the way arguments are passed to functions. It pushes them on the stack from left to right as they appear in the source code, whereas Full C compilers push them in the opposite direction. Under most circumstances this makes no difference. But, as it turns out, Full C compilers have a good reason for doing it "backward."

The functions **printf(), scanf(),** and their derivatives are written to accept any number of arguments. By passing arguments in the reverse direction, Full C compilers guarantee that the first argument will always be in a predictable position in the called function's stack frame—immediately beneath the return address. By having that argument indicate how many other arguments are being passed, the function will know how many arguments to process and where to find them—immediately beneath the first argument. (Refer to these functions in Chapter 12 to see how the first argument does this.)

Of course, Ken Thompson could have designed the first C compiler to push arguments in the "obvious" order and have these functions accept the control argument in the right-most position; that would have seemed unnatural to programmers, however, so he did the right thing by keeping the language natural and hiding the complexities within the compiler. When Ron Cain wrote the first Small C compiler, his run-time library did not include these functions and he did not envision it ever growing to that point; so compatibility in argument passing was not an objective.

At any rate, when I installed **printf()** and **scanf()** in the Small C library, I chose to leave the argument-passing algorithm unchanged and have the compiler pass an argument to the called function so it could locate the left-most argument. This is done in CL, the lower half of the CX register. The count can be obtained by calling the built-in function **CCARGC()** early in the called function, before CL gets changed. (See **printf()** and **scanf()** in files **FPRINTF.C** and **FSCANF.C,** respectively, in Appendix D for examples using **CCARGC().**)

What does all of this mean to programmers? It means that, except for one consideration, they can write portable programs even though they call functions that take a variable number of arguments. The exception is the order in which the arguments are evaluated. Since they are passed from left to right, they are also evaluated in that order. This means that argument expressions may affect the value of arguments to their right by performing assignments, increments, or decrements, whereas most Full C compilers would have the left-most arguments affected by those on their right. The best thing to do here is avoid writing function arguments with values that depend on the order in which arguments are evaluated. Also, when porting programs down to Small C, be on the lookout for this very subtle problem. We can always break down such argument expressions so that the assignments are performed by expressions which precede the function call. This problem was discussed under Argument Passing in Chapter 8.

Another difference surfaces when we wish to write a function that takes a variable number of arguments. With Small C, we must either place the control argument last, or call **CCARGC()** to locate it. In either case, the logic will have to be revised whenever the program is ported to another compiler.

Returned Values

Small C functions return only integer data types, whereas Full C compilers support functions that return any data type. Although this may seem restrictive, that is not the case.

For one thing, since characters are automatically promoted to integers wherever they appear in expressions, there is no practical difference between a function that returns an integer and one that returns a character that gets converted to

an integer. There is no limit on the type of value a Small C function might return. So a function might return a character with a statement like:

```
return (ch);
```

In such a case, the character gets promoted to an integer when the return expression is evaluated, rather than at the point of the call. Whatever the return expression yields, it will be a 16-bit value. The compiler doesn't really care whether it is a character, an integer, or a pointer. However, if the value of a function enters into operations with other operands, Small C will consider it to be an integer, whereas Full C compilers take returned values to be of the declared type.

Evaluation of Assignment Operands

Small C evaluates the left side of assignment operators before evaluating the right side. This means that variables (like subscripts) used in determining the destination of assigned values are not affected by the right side of the expression. Many Full C compilers, however, evaluate the right side first, thereby allowing the right side to influence the destination.

We should avoid writing expressions in which assignments, increments, decrements, or function calls on the right of an assignment operator affect objects that are used in determining the destination of the assignment.

Because they have been conditioned by other languages, most programmers would tend not to write expressions that violate this rule anyway. But it is possible in the C language, and the problems it can produce are particularly devious.

Octal Escape Sequences

Small C allows only the digits 0–7 in an octal escape sequence such as **'\127'.** Although this is consistent with Microsoft C and Turbo C, some Full C compilers also accept the digits 8 and 9, to which they give the octal values 10 and 11. Since Small C is more restrictive, this difference presents no upward portability problems. There could only be a problem when converting programs written for Full C compilers that accept this strange notation. And, even then, we would probably never see an octal number written in this manner.

Promoting Characters to Integers

Whenever we port a program from one C compiler to another, we must determine how the two compilers promote character variables to integers—with or without sign extension. A difference here can create problems that are very hard to debug. Fortunately, most C compilers (including Small C) do this the same way; unless specified otherwise, they treat characters as signed values.

Syntax of the #include Directive

Small C **#include** directives do not require quotation marks or angle brackets around the filename as Full C does. Most Full C compilers accept the angle brackets as an indication that the file is to be sought in a specific subdirectory (e.g., **\include**). Thus we usually see:

```
#include <stdio.h>
```

to include the standard I/O header file. On the other hand,

```
#include "Filename"
```

simply tells the compiler to look in the default directory. Small C accepts both forms, but treats them the same; it always looks in the default directory. For upward compatibility, we should always enclose **stdio.h** in angle brackets, and other files in quotation marks.

Old Style Assignment Operators

As with most modern C compilers, Small C does not recognize the original style of an assignment operator in which the equal sign was written as a prefix rather than a suffix. Therefore, sequences like =* and =& are taken as a pair of operators instead of a single assignment operator.

If there is a chance that we may port our programs to a compiler that accepts the old style assignment operators, we should write sequences like these with white space between the operators to avoid the ambiguity. We would probably do this anyway as a matter of good programming style.

File Descriptors

All Small C I/O functions use small integer values called *file descriptors* to identify files, whereas Full C compilers use both pointers and file descriptors, depending on whether high- or low-level functions are being used. This difference has no consequences as far as the logic of our programs is concerned, since file pointers and descriptors are normally used only to hold a value returned by an open function so that it can be passed to other functions. We might find a need to compare two file descriptors or pointers for equality or inequality, but here again there is no problem since we are comparing either values returned by an open function or a defined value like **stdin.** We do not really care what the actual values are or whether the variables are pointers or integers.

Full C compilers define in **STDIO.H** a file control structure called **FILE.** To declare a file pointer, the programmer writes something like:

```
FILE *fp;
```

Although Small C does not utilize file pointers, it supports the writing of this standard syntax by means of a trick. In its **STDIO.H,** Small C has:

```
#define FILE char
```

With this definition, the previous declaration defines **fp** to be a character pointer. It really doesn't matter what **fp** is declared to be since anything (integer, character, or pointer) is capable of holding a file descriptor. The main thing is that the declaration is compatible with Full C compilers.

Having declared file "pointers" in this way (or any other way for that matter), we are free to assign to them values returned by the open functions or any of the standard symbols **stdin, stdout, stderr, stdaux,** or **stdprn** (defined in **STDIO.H**).

Printf() and Scanf() Conversion Specifications

The Small C versions of **printf(), fprintf(), scanf(),** and **fscanf()** accept a binary conversion specification (designated by the letter b). Full C compilers do not support this feature, so its use must be considered nonportable.

Reserved Words

Keywords that are used in the C language are reserved; they cannot be used as identifiers in programs. Small C has a restricted set of reserved words, but full C compilers have more. To write upward compatible programs, we should avoid all of full C's reserved words, even though Small C may accept them. Following is the entire list of reserved words:

```
auto          double      *int        *switch
*break        *else       long        typedef
*case         entry       register    *unsigned
*char         enum        *return     union
const         *extern     short       *void
*continue     float       signed      volatile
*default      *for        *sizeof     *while
*do           *goto       static
              *if         struct
```

Words preceded with an asterisk are reserved in Small C. If we anticipate porting programs to the Microsoft or Turbo C compilers, we should also avoid the names:

```
cdecl     fortran     near
far       huge        pascal
```

Command-Line Arguments

C programs gain access to information in the command line that invokes the program through two arguments which are passed to **main()**. In Full C, we declare **main()** as:

```
main(argc, argv) int argc; char *argv[]; {
  ...
  }
```

when we want access to such information. **Argc** is an integer indicating how many argument strings are in the command line (including the program name and excluding redirection specifications). **Argv** is an array of character pointers, each pointing to a null-terminated argument string that has been extracted from the command line. By using **argv** to locate the strings and **argc** to know how many strings there are, we can write code that accesses the strings. The first argument string (pointed to by **argv[0]**) is supposed to be the program name. The others follow in the order of their appearance in the command line. Thus **argv[1]** points to the first string following the program name. Redirection specifications (see Chapter 15) are handled by MS-DOS and are not included as command line arguments.

Note: Since versions of MS-DOS earlier than 3.0 did not supply the program name, all C compilers provide a dummy value for the program name when running under older versions of MS-DOS. Small C substitutes an asterisk regardless of the version of the operating system.

Obviously, the above declaration for **main()** is not acceptable to Small C because it contains a declaration of a pointer array. So, with Small C, we declare:

```
main(argc, argv) int argc, *argv; {
  ...
  }
```

instead. While this declares **argv** to be a pointer to integers, it is in fact a pointer to an array of pointers (integer-sized objects) which locate the argument strings. By assigning each "integer" to a character pointer, it may then be used to access the designated string. To make all of this easier, the function **getarg()** is provided in the Small C library (see Chapter 12). This function takes the number of the argument sought, the address of a buffer in which to place it, the size of the destination buffer, **argc,** and **argv.** It locates the specified argument, copies it to the destination, and returns its length. If fewer command-line arguments exist than are necessary to supply the one specified, **getarg()** returns **EOF.**

When porting programs to Full C, it is probably best to port **getarg()** first, then simply change the declaration of **main()** to be compatible with Full C. We

might want to add **getarg()** to the standard run-time library of the new compiler so it will be there automatically when we need it. Of course, we could simply **#include** it into the source file of every program, but that requires keeping a copy of its source code available and lengthens compile times somewhat.

See Figure 15-1 and the surrounding text in Chapter 15 for a specific example of how command-line arguments are passed to programs.

CHAPTER 15

Executing Small C Programs

After a Small C program has been compiled, assembled, and linked, it ends up as an ordinary MS-DOS executable file with an EXE extension. Invoking *(executing)* such a program is just a matter of answering the MS-DOS prompt with the name of the program to be executed. However, four other matters of concern to the user are:

1. how to specify command-line arguments
2. how to redirect standard files
3. how to interrupt program execution
4. how programs indicate success or failure

Obviously, the way the program is written has the greatest influence on these questions, and so they cannot possibly be answered in general for all programs. They must be spelled out in detail in the program's documentation. But, there are certain traits which all programs produced by the same compiler have in common.

In this chapter, we look at the execution traits of Small C programs. Keep in mind that the Small C compiler is itself just another Small C program. So, whatever can be said of Small C programs in general applies equally to the compiler. We will have more to say about the compiler specifically in Chapter 16.

Command-Line Arguments

In Chapter 14 we studied the mechanism by which parameters can be passed to a program from the command line. Suffice it to say that whatever information

we add to the command line, after the program's name, can be viewed by the program and can influence the way it behaves.

Each string of visible characters (no spaces or tabs) is passed to the program as a separate argument. Arguments are passed in the order of their appearance. They can be separated by any amount of white space. A maximum of 19 arguments following the program name can be passed to the program. Additional arguments are ignored.

Redirection specifications (below) are invisible to the program. Arguments and redirection specifications can be mixed freely; no particular order is required (except that the program itself may impose an order on the arguments). The redirection specifications are extracted by MS-DOS, and what remains is passed on to the program.

With Small C it is not possible to include white space in an argument string. Other compilers (and the UNIX shell) let us enclose such an argument in quotation marks. However, Small C programs will split it into separate arguments and see the quotation marks as part of the first and last of these arguments.

Of course, it is up to the program to process arguments however it wishes. It may assign positional significance to arguments, or base their significance entirely on their values. It may even do both, assigning positional significance to some arguments (say the first one or two) and processing others based on their value alone. There are no rigid rules here—it is strictly a matter of choice.

Arguments that enable or disable program features (usually non-positional) are customarily called *switches*. Normal practice is to give switches a specific lead-in character—a hyphen (-) or slash (/), for example. UNIX programs use hyphens consistently. MS-DOS programs, on the other hand, use either or both. Microsoft has a preference for slashes, but many other developers follow the UNIX tradition. Still others straddle the fence by accepting both hyphens and slashes. Of course, switches are not required to have any lead-in character at all; but when non-switch arguments (like filenames) and switches must coexist, the lead-in is normally used to distinguish between them. Although this could be done on the basis of position, it is generally considered better style to let switches float in the command line—it is harder to remember where a switch must go

than to remember to put a hyphen in front of it.

Typically, switches consist of one or more alphabetic, numeric, or special characters. They usually have mnemonic value—what they do is suggested by their values, making them easy to remember. For example, the switch:

```
-ep105
```

might tell a text formatter to "end on page 105." The command:

```
prog -x zz <abc /1234
```

would invoke **PROG.EXE** passing **-x, zz,** and **/1234** as arguments. The string **<abc** is not passed since MS-DOS takes it as a redirection specification (below) and drops it from the command line before passing it to the program. When **PROG** receives control in **main(),** its stack will be set up as indicated in Figure 15-1.

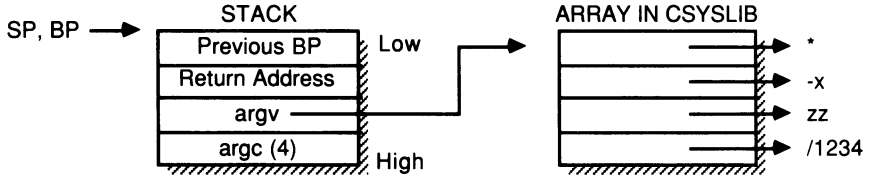


Figure 15-1. Arguments Passed to Small C Programs

As we saw in Chapter 14, **argc** and **argv** are declared as formal arguments in **main()** and therefore can be accessed by name. The function **getarg()** in the Small C library (Chapter 12) is designed to help fetch the command-line arguments.

Input/Output Redirection

The general concept of standard files and redirection was covered in Chapter 12 where we discussed them from the viewpoint of the programmer. Now we consider them from the user's perspective. Current versions of MS-DOS completely handle standard file assignments, so the following discussion pertains universally to programs that work with standard files under MS-DOS, not just Small C programs.

Of the five standard files that are open and ready for use when a program begins execution, two are redirectable—the *standard input* (**stdin**) and *standard output* (**stdout**) files. Unless specified otherwise in the command line, **stdin** is assigned to the keyboard and **stdout** is assigned to the screen.

Failure to realize this can create confusion, because a program waits for user response when it inputs from the keyboard. Many programs written in the UNIX tradition do not prompt for input when **stdin** has its default assignment. So novice users may find themselves waiting for the program while the program is waiting for them. If they do not realize that the program wants keyboard input, they will think the program is hung up.

The important thing to remember about entering a file from the keyboard is how to signal that the end of the file has been reached. This is done by entering a <control-Z>, the standard end-of-file sentinel for MS-DOS. As we saw in Chapter 12, the end of a "cooked" disk file is signaled either by a <control-Z> or the physical end of the file. With the keyboard, however, there is no physical end of the file, so a <control-Z> must be used. Depending on whether the program is reading the file on a line-by-line or a character-by-character basis, the user may or may not have to hit the ENTER key after the <control-Z>.

By placing *redirection specifications* in the command line, the user causes the default assignments for **stdin** and/or **stdout** to be changed for the present execution of the program. Redirection specifications can be placed anywhere in the command line after the program name. To redirect **stdin** we enter a < followed by a file specification. When that is done, input to **stdin** comes from the designated file or device instead of the keyboard. A > followed by a file specification redirects **stdout** to the indicated file or device. This causes output through **stdout**

to go to the designated file or device instead of the screen.

File specifications may or may not include drive, path, and extension. The filename itself is required (except for devices like **COM1:**), and if the indicated file has an extension, that too must be given. Omitting the drive designator directs MS-DOS to the default drive, and omitting the path assigns the reference to the current subdirectory. MS-DOS's special device file names (**AUX, PRN,** etc.) and actual device names (e.g., **LPT1:**) are acceptable as file specifications.

When **stdout** is redirected to a disk file, and the named file does not exist, a new file with the specified name is created. If the file does exist, then it is overwritten from the beginning. Standard output can be concatenated to an existing file by writing the redirection specification with **>>** instead of **>.** If the named file does not exist, however, this has the same effect as an ordinary output redirection—a new file is created.

Programs that simply read data from **stdin,** process it, and write the result to **stdout** are often called *filters.* Table 15-1 illustrates several examples of redirecting the input and output of such a filter program.

Command	Comment
`filter <abc`	Filter file **abc** to the screen.
`filter <abc >def`	Filter file **abc** to file **def.**
`filter <abc.x >>def`	Extend file **def** with filtered **abc.x.**
`filter >prn`	Filter keyboard input to the printer.

Table 15-1. Redirecting Standard Input and Output Files

Interrupting Program Execution

As data spews onto the screen from **stdout** or **stderr,** the user may wish to freeze the output long enough to study something before it scrolls off the screen. This can usually be done by entering a <control-S> from the keyboard. This keystroke is not taken as data for **stdin,** but simply pauses the program instead. A second keystroke resumes program execution. This character, too, is dropped from the input file.

Many programs are also sensitive to the <control-C> or <control-BREAK> keystrokes. If MS-DOS has not been told otherwise, when it sees one of these keystrokes it cancels the program. Even if MS-DOS is ignoring them, the Small C **get** functions, when inputting from the keyboard, take <control-C> as a signal to abort the program. Of course, we can use the **read** functions, which do not cook the data, to override this action. Also, **poll()** can be used to poll the keyboard with or without sensitivity to <control-C> and <control-S> keystrokes.

Exit Codes

A Small C program may terminate execution in four ways. It may allow control to reach the end of **main(),** at which point control returns to MS-DOS. It may call either **exit()** or **abort(),** or it may be cancelled by the user.

The functions **exit()** and **abort()** are really the same function with two names. They take a single integer argument—an error code. If the error code is zero, a normal exit is indicated and control returns quietly to MS-DOS with an exit code of zero. However, if the error code is not zero, the program displays:

```
Exit Code: n
```

(where ***n*** is the error code) and returns to MS-DOS passing it the error code. The value passed to MS-DOS can be tested by means of the batch command:

```
IF ERRORLEVEL ...
```

When a program aborts because of insufficient memory, it issues an exit code of 1. This condition is caught by **avail(),** which is called whenever memory is allocated by **malloc()** or **calloc(),** or whenever a program calls it directly.

When **poll()** or any of the **get** functions detects a <control-C>, they abort the program with an exit code of 2. When control is allowed to reach the end of **main(),** an implicit:

```
exit(0);
```

is issued to close open files and return to MS-DOS.

CHAPTER 16

Compiling Small C Programs

The Program Translation Process

The output of the Small C compiler is designed to be compatible with the Microsoft and IBM assemblers. As of this writing, Small C does not have an assembler of its own, although one is being developed and may even be available as you read this. The new assembler is to be fully documented in a separate book. The following comments are intended to provide helpful information about the overall program translation process, regardless of the assembler used. Refer to the assembler's documentation for details about the assembly process. Also refer to the MS-DOS documentation for specific information about the linker.

As we noted, the Small C compiler generates assembly language output. This makes compiling a Small C program a three step process. First the program is passed through the compiler to produce the assembly code (usually in a file with an **ASM** extension). Next, it goes through an assembler to create a linkable object file (**OBJ** extension). And finally, it is translated by the linker into an executable file (**EXE** extension). Of course, the entire process may be captured in a batch file so that it becomes a single step for the user.

The complete process is illustrated in Figure 16-1, which shows the general case where a multi-part program is compiled and assembled in parts, then linked together with modules from several libraries.

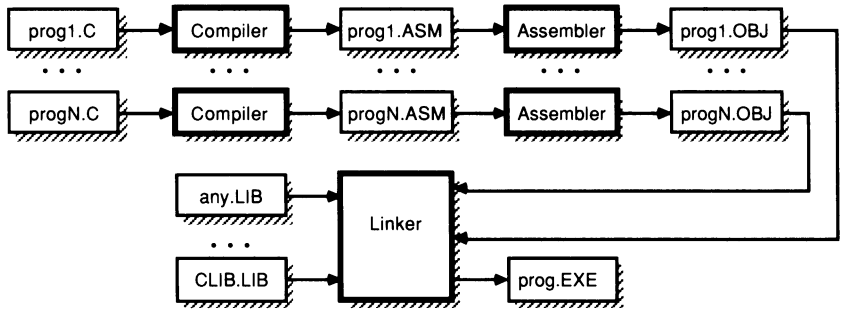


Figure 16-1. Compiling Small C Programs

The **ASM** file produced by the compiler is an ordinary ASCII file, so it can be typed on the screen, printed, or edited. The code generated by the compiler is plainly visible, making it easy to see what the compiler did.

The Assembler

The assembler primarily does three things. It translates the mnemonic operation codes to numeric codes that are meaningful to the CPU. It converts label references to actual addresses. And it casts the result into the **OBJ** file format that the linker requires. As far as we are concerned here, the assembler is just a tool that must be applied to the output of the compiler. Since the compiler never generates code that produces assembler errors, there is no need to consider error conditions. Refer to the assembler documentation for specific information about invoking the assembler.

The Linker

DOS comes with a linker which is compatible with the one that comes with the Microsoft and IBM assemblers. And since the Small C assembler will use this linker, we can be more specific about this phase of the process.

As Figure 16-1 shows, the linker is the hub of the process; it brings everything together to produce the executable file. Besides concatenating together the various modules of the program, its main function is to *resolve* external references. When something is declared **extern** in one module, and is defined globally in another, the linker locates where it is defined (the *entry point*) and patches every reference with its actual address. This cannot be done by the assembler because when it assembles modules, it has no idea where external targets are defined or what addresses they will have when the modules are combined.

Two errors are likely to occur when linking. Some external references may not be resolvable because there is no definition in any of the modules. This condition produces a list of the offending names and the modules from which the references are made. The list is introduced by the message:

```
Unresolved Externals:
```

Also, two or more modules may contain global definitions with the same name, creating ambiguity. The linker complains about this condition with the message:

```
Symbol Defined More Than Once:
```

This is followed by the name of an offending definition.

The linker can be pointed to libraries (**LIB** extension) as well as object files. A library is a file that contains a collection of object modules that are copied into it by means of a library manager. The linker is designed to search libraries for modules with entry points that match unresolved references. When such a module is found, it is copied from the library into the program. This fixes the addresses of its entry points so the linker can then resolve references to them. With this arrangement, only those modules in the library which are actually needed are added to the program.

Notice in Figure 16-1 that more than one library can be scanned. When this is done, the linker scans the libraries in the order in which they are named to the linker. Once it has moved from one library to the next, it does not go back to the previous library. This means that when a module in the second library contains

external references to a module in the first library, an unresolved reference will occur. This can always be avoided by specifying the standard library (Small C's **CLIB.LIB**) last.

Other libraries may contain special purpose functions (the windowing functions of the Small-Windows library, for instance). Such functions are likely to call the standard functions. If the standard library is processed first, then these references to standard functions cannot be resolved. Therefore, it is important to *always specify the standard library last.*

Even within a single library, the order of the modules may be such that backward references between modules exist. This does, in fact, happen. But the linker is designed to resolve all references to modules in the current library before moving to the next one.

The Compiler

The Small C compiler has the filename **CC.EXE.** It uses standard file redirection and command-line arguments (see Chapter 15) to determine where to obtain its input, where to send its output, and which run-time options to apply.

The Small C compiler employs a flexible algorithm for determining its input and output files. The purpose is to make it equally convenient for production compiles and special cases. By default, Small C obtains its input from **stdin** and sends its output to **stdout.** Therefore, when it is invoked by the command:

```
CC
```

it simply displays its **signon** message and waits for keyboard input. It's response to whatever it receives is shown immediately on the screen. This mode of operation, which makes studying the compiler's behavior very convenient, is unique to the Small C compiler. Whenever we wish to see what the compiler will generate for a given situation, all we have to do is execute it in this manner, query it with a program fragment, and observe its response. If we want a printed record of the session, we can redirect the output, as with:

```
CC >PRN
```

We can also have it include the source lines with its output so it will be clear which source lines produced which assembly code. This is done with:

```
CC >PRN -L1
```

The letter L does not have to be uppercase. It is shown that way only to avoid being confused with the numeric digit 1. More will be said about the *listing* switch below.

Of course, other possibilities exist for the default use of the standard input and output files. For instance, if we wish to see what the compiler does with an actual program, we could invoke it with:

```
CC <PROG.C -L1
```

which would compile the program to the screen together with its source code. We could then use <control-S> keystrokes to alternately pause and resume execution as the screen scrolls. Combining the last two examples would make the output go to the printer.

When we use the listing switch in this way, the source lines in the output are each preceded by a semicolon. The purpose is to make them appear as comments to the assembler. Therefore, mixing the source listing with the output does not create assembler errors. We might invoke the compiler with:

```
CC <PROG.C >PROG.ASM -L1
```

to create a file for the assembler—a file which could also be viewed on the screen or printed.

Using the standard files gives the compiler a great deal of flexibility. But for production compiling we should not have to enter two redirection specifications complete with filename extensions. That would be a bit unwieldy. So, in those situations where we simply want to compile a program the simplest way possible, we can enter:

```
CC PROG
```

The compiler will take the non-switch argument **PROG** as a filename and assume that it should be used with different extensions for both input and output. It will assume an extension of **C** for the input and **ASM** for the output. So, in this case, it would input **PROG.C** and output **PROG.ASM.** No listing will be generated either on the screen, on the printer, or in the output. If more than one filename is given, the compiler will concatenate them on input and apply the first name to the output. Thus,

```
CC PROG PROG2 PROG3
```

will input **PROG.C, PROG2.C,** then **PROG3.C** as though they were a single input file. Output will go to **PROG.ASM.**

Finally, we can use output redirection together with named files. This allows us to override the assumed output filename. For example,

```
CC PROG PROG2 >PRN
```

would compile **PROG.C** and **PROG2.C** to the printer, and:

```
CC PROG PROG2 >OUT.A
```

would compile them to a file named **OUT.A.** Note that redirection specifications do not have default filename extensions.

While the compiler is executing, it may be interrupted in either of two ways. It will pause on a <control-S> keystroke and continue on the next keystroke of any type. Also, a <control-C> aborts execution with an exit code of 2. The switches which control compiler behavior are:

1. The **-M** switch lets us *monitor* progress by having the compiler write each function header to the screen. This also helps isolate errors to the functions containing them.
2. The **-A** switch causes the *alarm* to sound whenever an error is reported.
3. The **-P** switch causes the compiler to *pause* after reporting each error. An ENTER (carriage return) keystroke resumes execution.
4. The **-L#,** switch calls for a source *listing*. The # represents a single

numeric digit which specifies the file descriptor of a standard file which is to receive the listing. Table 12-1 lists these values. Zero should not be used because it specifies an input file. As we saw above, 1 specifies the standard output file. In that case, since the output may also go to the same file, a semicolon precedes each line of the listing, making it appear to the assembler as a comment. Specifying **-L2** sends the listing to the standard error file which is assigned to the screen and cannot be redirected.

5. The **-NO** switch specifies *no optimizing*. We can use this switch to see the raw, unoptimized code.

6. The null switch, or any unrecognizable switch, causes the compiler to abort after displaying the help line:

```
USAGE: CC [FILE]... [-M] [-A] [-P] [-L#] [-NO]
```

This little bit of assistance is usually enough to remind us of what we need to know. The brackets designate their contents as optional, and the ellipsis means that more files can be specified.

Both uppercase and lowercase letters are acceptable in filenames and switches. Table 16-1 illustrates sample commands which invoke the compiler, and describes their effects.

Command	Comment
`cc -no`	Compile from the keyboard to the screen without optimizing.
`cc <prog.c -l1 -p`	Compile **prog.c** to the screen, list the source as comments in the output, and pause on errors.
`cc <abc.c >xyz.a -m`	Compile **abc.c** to **xyz.a** while monitoring function headers on the screen.
`cc abc def -a`	Compile **abc.c** then **def.c** to **abc.asm,** sounding the alarm on errors.

Table 16-1. Invoking the Compiler

CHAPTER 17

Compiling the Compiler

How to Compile the Compiler

Since the Small C compiler is really just another Small C program, we follow the steps in Chapter 16 to compile it just as we would any other program. The only difference is the number of source files and their names. Whereas Figure 16-1 illustrates the general procedure, Figure 17-1 is specific to the compiler itself.

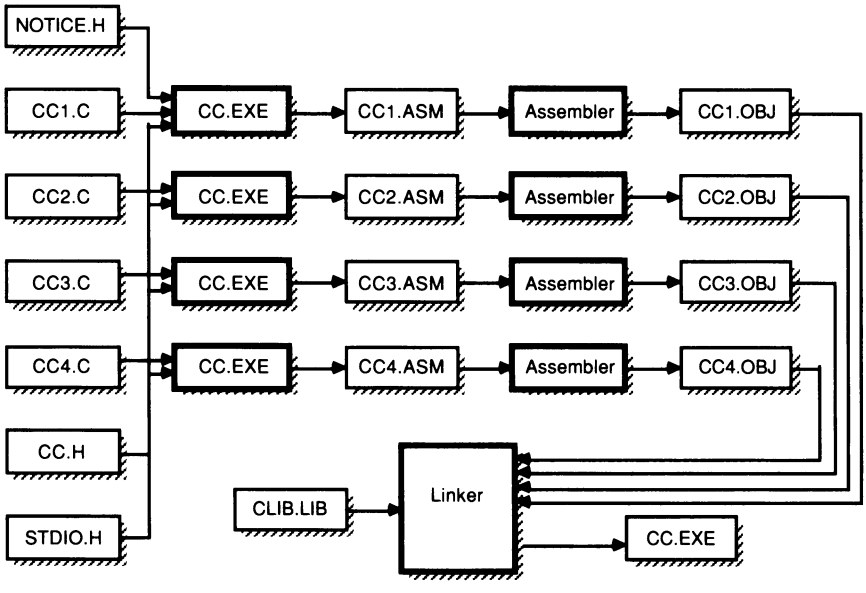


Figure 17-1. Compiling the Compiler

The compiler is organized into four parts, each of which is compiled and assembled separately. The **OBJ** files are then combined by the linker with modules from the library **CLIB.LIB** to produce the new **CC.EXE** file. Each source file includes two header files, **STDIO.H** and **CC.H.** In addition, the first source file includes **NOTICE.H,** which contains the **signon** notice. Listing 17-1 shows a batch file that will perform the entire operation. This file uses the Microsoft assembler. It assumes that we have sufficient disk space and that the PATH environment variable is properly set so that the assembler and linker can be found by MS-DOS. If this is not so, then adjustments will have to made accordingly.

```
cc  cc1 -m -a -p
masm cc1;
del cc1.asm
cc  cc2 -m -a -p
masm cc2;
del cc2.asm
cc  cc3 -m -a -p
masm cc3;
del cc3.asm
cc  cc4 -m -a -p
masm cc4;
del cc4.asm
link cc1 cc2 cc3 cc4,cc,cc,clib.lib
```

Listing 17-1. Batch File to Compile the Compiler

The four parts of the compiler are divided functionally so that changes to one part will not normally require changes to the other parts. Having compiled the entire compiler once and having kept the **OBJ** files, it is then only necessary to compile and assemble the parts that are actually affected by a change. Notice that the batch file in Listing 17-1 deletes the **ASM** files, but retains the **OBJ** files for just that reason. Listing 17-2 shows a batch file for recompiling just one part of the compiler.

```
cc   cc%1 -m -a -p
masm cc%1;
del  cc%1.asm
pause CONTROL-C TO SKIP THE LINK STEP
link cc1 cc2 cc3 cc4,cc,cc,clib.lib
```

Listing 17-2. Batch File to Recompile One Part of the Compiler

Notice that the link step is optional in case two or three parts must be recompiled. We can perform the link step only with the last part. The **%1** in this file stands for the first command line argument. If the batch file were named **CCC.BAT,** then to invoke it for part three of the compiler we would enter

```
CCC 3
```

General Advice

CC.H contains **#define** statements that pertain to the compiler as a whole. Changes to this file usually require that the entire compiler be recompiled. However, by keeping track of which symbols in **CC.H** have changed, we can use a text editor to perform a global search on each of the source files to determine which ones must be recompiled.

The symbol **DISOPT** is contained in a **#define** directive in **CC4.C** where the code generation and optimizing logic resides. Currently the directive is contained in a comment so that it will be ignored by the compiler. However, if we remove the comment delimiters and recompile **CC4.C,** the new compiler will list on **stdout** the frequencies of the optimization cases it applies. This is handy when working with the optimizer; it helps us to decide whether or not a given optimization is likely to be worth its cost in performance and compiler size. There is no point in having the optimizer look for cases that seldom arise.

Notice in Figure 17-1 that the new executable compiler **CC.EXE** replaces the previous one. This has ramifications. What is the probability that the new compiler will work properly? Close to zero, no doubt. So we have just replaced our production compiler with something containing bugs. Failure to realize this, when we recompile the compiler to fix it, leads to interesting results. The buggy

compiler may run without a complaint. And the new compiler may also link properly. But when we run the new compiler, look out. Chances are that it will go berserk—that is, if our bug caused bad code to be generated. If we are on our toes, we will recognize the symptoms and realize what happened. But the symptoms of many compiler bugs are not obvious, and some are not even predictable.

Of course, we have a copy of the original, unaltered compiler somewhere, do we not? Sure we do. So we can restore **CC.EXE** from that copy and proceed to recompile our fixed compiler properly. To avoid the loss of time, we will probably want to keep a reliable copy of the compiler handy and make sure to use it for each new compilation. Eventually, when we are convinced of the dependability of our revised compiler, we will want to begin using it as our production compiler so future versions of the compiler can incorporate new features that are supported by the revised compiler.

Now, suppose we add support for some new constructs. We place the new compiler in production status, and proceed to use the new constructs in future revisions of the compiler. By making the source code dependent on the enhanced compiler, we are committing to its reliability. If we should get down the road a way, and discover that our production compiler has problems, then falling back to the previous production compiler will not work because the old compiler will choke on the new constructs which we have put in the source files. At that point we will have to resort to some messy patching of the source files to make them acceptable to the old compiler. This will probably involve commenting out logic that implements enhancements and/or rewriting the new constructs in the old way. We must then fix the previously undiscovered bug, recompile, test, reinstall this as the production compiler, remove the temporary patches, and recompile again. Finally, we will be back where we were. The point is that time saved in testing may well be spent in even more disagreeable ways.

Conclusion

At this point, we know the Small C language, its repertoire of library functions, how to use the compiler, and how to compile new versions of the compiler. Now we come to the interesting part—what goes on inside the compiler.

Part 3

Inside the Small C Compiler

PART 3

Inside the Small C Compiler

Much of the appeal of the Small C compiler lies in the fact that it holds no secrets. Everything is visible to those who want to know. This part of the book opens the compiler to reveal its inner workings. While not necessary for using the compiler, knowledge of what the compiler does and how it does it can improve our appreciation of the language and our use of it.

This material is in places hard to grasp. However, I have tried to simplify it as much as possible, especially in the most difficult areas. Numerous figures, tables, and special listings are provided to help clarify the concepts.

As a prerequisite to this study, you should be familiar with the Small C language as described in Part 1. In addition, you should have at least a rudimentary knowledge of assembly language programming for the 80x86 processors. If you are weak in this area, you will probably find that you can manage by relying on the explanations found in the text. Nevertheless, it would be easier with some foreknowledge of the 8086 instruction set.

Complete listings of the compiler are contained in Appendix C. You will need to study these listings as you read the explanations of the compiler's logic. Appendix H has been provided to help you find the compiler's functions. It lists the functions alphabetically with each function's source file and page in Appendix C.

In the difficult parts of the expression analyzer, pseudo-code listings are provided. These stand midway between the text and the actual compiler listings. The text explains the pseudo-code. After that, you can easily relate the pseudo-code to the compiler listings, since they correspond almost line for line.

I am sure that you will find your exploration of the Small C compiler a rewarding experience. No doubt you will gain a feeling of satisfaction and self confidence at having discovered the secrets of a real compiler.

Organization of the Compiler

As indicated in Figure P3-1, the Small C compiler is essentially a *parser* with subordinate *front end* and *back end* functions. The front end reads, preprocesses, and scans source code for the parser, while the back end expresses the outcome in assembly language. Basically, the front end is the input side of the parser and the back end is the output side.

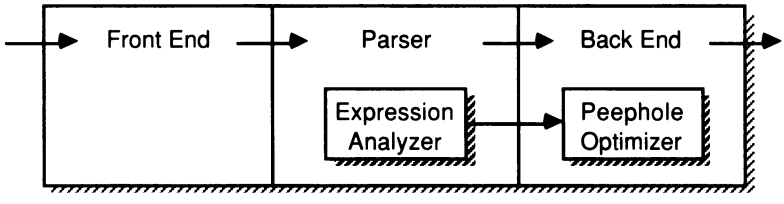


Figure P3-1. Organization of the Small C Compiler

It would be nice to simply start at the top and work our way down through the underlying functions. But we would certainly bog down in the middle as we became overwhelmed with new functions while trying to remember where we were in the parsing process. So instead, we shall first move from the bottom up and then from the top down, connecting in the middle. This approach requires some patience at first, because the overall picture does not form until the last stage. However, it does make the top down stage much easier, since we encounter only functions that are familiar to us.

First, before looking at the compiler itself, the routines in the **CALL** module of the library are examined. We do this since these routines are called frequently

by the generated code, and so a knowledge of them is essential to an understanding of the output of the compiler.

Next, samples of the code generated by the compiler are compared to the source statements that produce them. Comparing the compiler's input and output gives us a feel for what the compiler has to do and provides a basis for understanding the back end of the compiler.

After that, the data structures of the compiler are examined. These include the symbol tables (global and local), the **switch** table, the **while** queue, the literal pool, the staging buffer (used for optimizing expressions), and the macro buffers (used with **#define** commands).

Then, the back end of the compiler is explored to learn the functions that directly produce the output. Next, to complete the bottom up phase, the front end of the compiler is studied.

With the preliminaries out of the way, we are finally be ready to tie it all together by studying **main()** and the parsing functions. In so doing, we move from the top down, through the heart of the compiler.

Since expression analysis is such a large part of parsing, and since it is easily separated from the rest of the compiler, we cover it in a separate chapter. Likewise, the code optimizer is treated in a chapter of its own.

Finally, since Small C is meant to be experimented with, suggestions for further development are given. These projects, which range in difficulty from very easy to very hard, are a rich source of ideas for student assignments in courses on compiler construction.

CHAPTER 18

The Call Routine

The term **CALL** is an acronym for *C Arithmetic and Logical Library,* the original name for this collection of routines. Actually, however, the **CALL** routines are not a library, but a single module of routines in the library; they no longer contain arithmetic routines since the 80x86 processors supports all four basic math operations directly. In addition, two of the three types of routines in **CALL** do not even fit the description. Nevertheless, the name has stuck. So the term **CALL** is really better understood not as an acronym, but as the word "call," referring to the fact that these routines are called in lieu of in-line code. The **CALL** module is always linked to every Small C program.

See Appendix D for a listing of the **CALL** module. Since the **CALL** module is written in assembly language, it might be helpful to review the 80x86 CPU architecture in Appendix B before proceeding. Also keep in mind that the mapping of Small C registers to 80x86 registers is:

Small C	**8086**
primary	AX
secondary	BX
stack pointer	SP

The **CALL** module breaks down into four parts: the start-up routine, a short routine for fetching the number of arguments passed to a function, the logical comparison routines, and the **switch** evaluation routine.

The Start-Up Routine

The first section of code in **CALL** begins with the label **start,** which is designated by the **end** directive that terminates **CALL,** as the initial entry point for the program. This is the point at which every Small C program receives control from the operating system. Since this is the only **end** directive (of all the modules in a program) that specifies an initial entry point, there is no ambiguity. When writing modules in assembly language which are to be linked with Small C (or any other language) programs, be sure not to specify an initial entry point with the **end** instruction. One look at the output of the compiler should convince us that the compiler also adheres to this rule.

The start-up routine serves four purposes: it sets up the CPU's memory segmentation registers, it sets up the memory heap, it sets up the stack, and it jumps to the function **_main()** in the library module **CSYSLIB.** That function then parses the command line arguments, and calls **main()** in the program. With this arrangement, if control reaches the end of **main()** it returns back to **_main()** in **CSYSLIB** where the statement:

```
exit(0);
```

is waiting to terminate the program gracefully. See Figure 5-1 for a diagram of the memory arrangement which this routine sets up.

The first thing **start** does is to set the data segment register (DS) with the paragraph address of the data segment (where the global objects reside). Note: A paragraph is a 16-byte unit of memory that begins on a 16-byte boundary.

This ensures that the globals will be referenced properly. Next, it subtracts this address from the amount of memory that is currently available to the program to derive the amount of memory in paragraphs that can be used for the heap and the stack. If more than 64K bytes remain, then only 64K bytes are used; if less, then whatever remains is used. This number, the length in paragraphs of the combined data/heap/stack segment is converted to a byte offset to arrive at the value which is placed in the stack pointer (SP). This makes the stack begin at the high end of the segment. Finally, the stack segment register (SS) is set to the same value as DS.

Next, a check is made to ensure that there is enough memory for a minimum stack of 256 bytes. If that should fail, there is not enough memory for the program to even start, so the program aborts with an exit code of 1.

If there is enough memory, however, the global integer **_memptr** in **CSYSLIB** is set to the address (offset) of the beginning of the heap. This is where the memory allocation routines keep track of the end of the heap,which is the beginning of free memory. If this should ever exceed SP, a stack overflow occurs.

Last of all, a jump to **_main()** is executed to continue the start up process in **CSYSLIB.**

Three instructions are commented out at the end of **start.** If they were enabled, they would return to DOS the part of available memory that is not being used by the program. If they are used, however, the MS-DOS **DEBUG** utility will not work, so they have to be deactivated.

The Argument Count Routine

_Ccargc is the routine (function), referred to in Chapter 8, that returns a count of the number of arguments passed to a function. Small C places this count in the CL register before calling a function. The called function can get the count by calling **ccargc** before doing anything else. There are only three steps to this routine—move CL to AL, set AH to zero, and return to the caller. The number of arguments passed to the function is then returned in the primary register (AX), where functions normally return their values.

Logical Routines

Ten logical comparison routines reside in the **CALL** module. They are:

Label	Operation
__eq	equal to
__ne	not equal to
__lt	less than
__le	less than or equal to
__ge	greater than or equal to
__gt	greater than
__ult	unsigned less than
__ule	unsigned less than or equal to
__uge	unsigned greater than or equal to
__ugt	unsigned greater than

Except for the type of comparison made, each routine operates in the same way. It compares BX (left operand) to AX (right operand), and returns 1 if the condition is true, or zero if it is false.

A macro instruction called **compare** is defined and then expanded to produce each routine.

An additional routine **__lneg** performs the logical negation of the value in AX. If it is *true* (non-zero), *false* (zero) is returned; otherwise, *true* (one) is returned.

By implementing these operations as subroutines, rather than as in-line code, several bytes per call are saved. This is a size-over-speed compromise.

The Switch Evaluation Routine

__Switch is the routine that evaluates **switch** conditions. Having a routine for this purpose reduces the size of programs that use **switch** statements with large numbers of **cases,** since each **case** does not require in-line code to compare the **case** value to the **switch** expression.

See Chapter 10 for details on the operation of **switch** statements. When Small C encounters a **switch** statement, it generates code to evaluate the **switch** expression and place its value in AX. This is followed by a **jump** around the code which is associated with the various case prefixes. At that point there is a call to **__switch** followed by a table of address/value pairs. Each pair gives the address of the code for a **case** together with the case's value. The table is terminated with a zero address. Next, if a **default** prefix exists, there is a **jump** instruction to the **default** address. Finally, all of this is followed by whatever code comes next in the program. Figure 18-1 illustrates the arrangement.

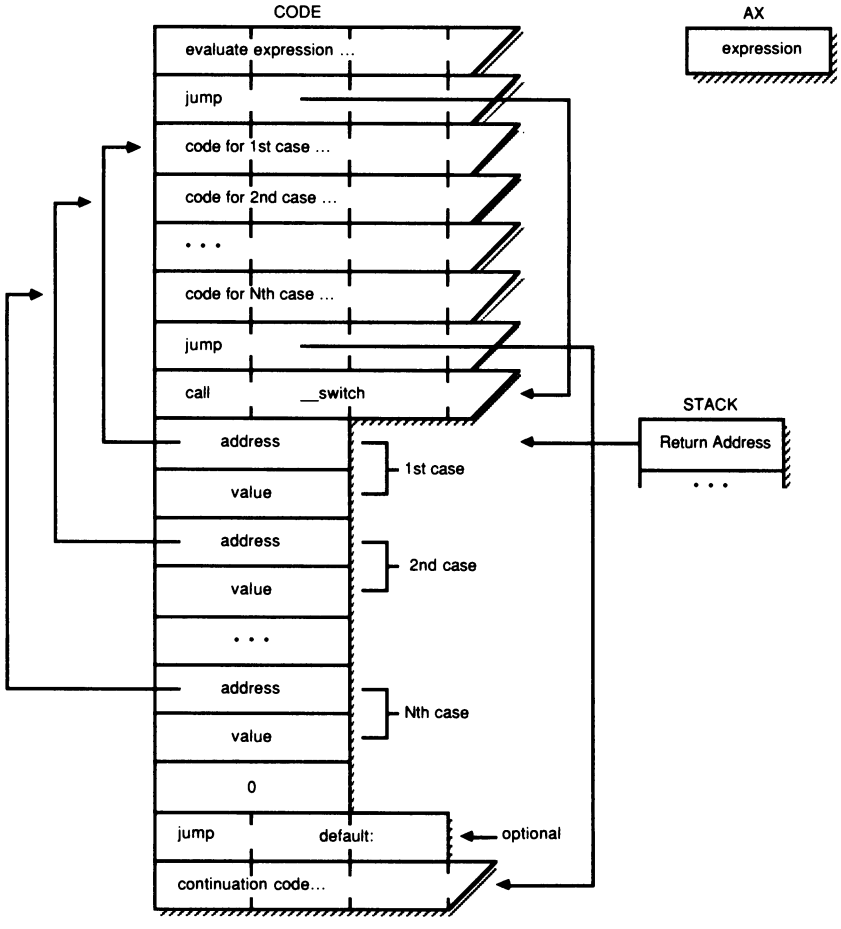


Figure 18-1. Set-up When __Switch is Called

Notice that the return address for the call to **__switch** is really the address of the **switch** table, not really the address to which control should return; in fact, control should never get there. **__Switch** will see to it that control goes to one of the **case** addresses, the **default** address, or the continuation code.

__Switch pops the table address into BX. This adjusts the stack properly for the jump that transfers control back into the program; that is, the **pop...jump** sequence is effectively a return. But in this case, one of a number of possible locations is chosen.

Next, control enters a loop in which:

1. The address in the current table entry (pointed to by BX) is moved to CX where it is tested for zero, signalling the end of the table. In that case, a **jump** instruction takes control to the address in BX (plus 2); that is, to whatever follows the table. If a **default** prefix exists, then this location holds a **jump** to the default address. Otherwise, this is the beginning of whatever code follows the **switch** statement.

2. The **case** value at BX+2 is compared to the value of the expression in AX. If they match, a **jump** to the address in CX transfers control to the code for the matching **case** prefix. However, if they do not match, the current table address in BX is incremented by 4 (to the next address/value pair) and control goes back to step 1.

One way or another, control reenters the program. Either it goes to a matching **case** prefix, the **default** prefix, or if there is no **default**, to whatever follows the **switch** statement.

CHAPTER 19

Generated Code

In this chapter, a number of program fragments are presented together, with the assembly language code they produce. The idea is to become familiar with what the compiler does before seeing how it does it.

As seen in Chapter 16, the compiler can be invoked in such a way that we can enter source statements from the keyboard and observe on the screen the code which they generate. Therefore, any of these examples can be verified.

Before proceeding, we need to establish some basic concepts underlying the compiler's view of the CPU. The compiler sees the CPU as a pair of 16-bit registers—*primary* and *secondary*—and a *stack pointer.*

The primary register is the recipient of expression values. When a *unary* operation is performed, the operand is placed in the primary register, and the operation is performed. When a *binary* operation is performed, the left-hand operand is evaluated first. Then, when it is seen that a binary operator follows, it is pushed onto the stack while the right-hand operand is evaluated—also in the primary register. After that, the left operand is popped into the secondary register and the operation is performed, with the result going to the primary register. If possible, the optimizer eliminates this use of the stack by moving the left-hand operand directly to the secondary register (if it is not needed for evaluating the right operand) or by generating the left operand directly in the secondary register (at the point where the **pop** would be placed).

As it employs the 8086 CPU (Appendix B), the compiler uses the AX register for the primary register and BX for the secondary register. Both of these registers consist of a pair of 8-bit registers. AX, for example, consists of AH (the high-order byte) and AL (the low-order byte).

In the following examples, the compiler's code optimizer is allowed to operate as usual. So the assembly code you see is the normal, optimized code. You may find it instructive to try the **-NO** switch on these examples.

Constants

The examples in Listing 19-1 show two functions, each containing constant expressions as stand-alone statements. Each statement generates a:

```
MOV AX,...
```

instruction that moves the constant value to the primary register. Notice also that a constant expression like:

```
123+321
```

is evaluated by the compiler and only the result is loaded. Since the expression yields a constant, it is evaluated at compile time rather than run time. The advantage should be obvious. Next, notice the handling of character strings. Each string generates:

```
MOV AX,OFFSET _m+n
```

where ***m*** is the number of a compiler-generated label and ***n*** is an offset from the label to the beginning of the string. The operator **OFFSET** tells the assembler to place the offset part of the address in AX rather than the operand at that address. Since Small C uses a small memory model, the offset is effectively the full address. Thus, the presence of a character string produces the address of the specified string in the program. All of the strings in a function are dumped into the data segment, one after the other, when the compiler finds the end of the function. This *literal pool* for the function is preceded by a single compiler-generated label. The underscore character is acceptable in labels. Its presence before the label number makes the assembler see the number as part of a label name rather than a number. Notice that each string is terminated with a null byte in the standard C fashion.

Input	Output

`func1() {`	`CODE SEGMENT PUBLIC`
	`ASSUME CS:CODE, SS:DATA, DS:DATA`
	`PUBLIC _FUNC1`
	`_FUNC1:`
	`PUSH BP`
	`MOV BP,SP`

`123;`	`MOV AX,123`
`123 + 321;`	`MOV AX,444`
`"abc";`	`MOV AX,OFFSET _1+0`
`"def";`	`MOV AX,OFFSET _1+4`

`}`	`POP BP`
	`RET`
	`CODE ENDS`
	`DATA SEGMENT PUBLIC`
	`_1 DB 97,98,99,0,100,101,102,0`
	`DATA ENDS`

`func2() {`	`CODE SEGMENT PUBLIC`
	`ASSUME CS:CODE, SS:DATA, DS:DATA`
	`PUBLIC _FUNC2`
	`_FUNC2:`
	`PUSH BP`
	`MOV BP,SP`

`'a';`	`MOV AX,97`
`'\1\1';`	`MOV AX,257`
`"ghi";`	`MOV AX,OFFSET _2+0`

`}`	`POP BP`
	`RET`
	`CODE ENDS`
	`DATA SEGMENT PUBLIC`
	`_2 DB 103,104,105,0`
	`DATA ENDS`

Listing 19-1. Code Generated by Constant Expressions

Segment directives of the form:

```
name SEGMENT PUBLIC
```

tell the assembler to assemble the following code into the named segment. Small C works with just two segments—**CODE** and **DATA.** The assembler is switched between these segments by closing the current segment with:

```
name ENDS
```

and opening the other one with another segment directive. The result is a collection of segment fragments bearing the names **CODE** and **DATA.** The linker combines the fragments with the same name into a single segment in the **EXE** file. The directives:

```
ASSUME CS:CODE, SS:DATA, DS:DATA
```

tell the assembler what to expect in the segment registers at execution time. The start-up routine in the **CALL** module of the library sees to it that the segment registers contain these values.

Finally, in the second function, notice that character constants result in just their numeric values.

Global Definitions

The examples in Listing 19-2 illustrate the code generated when global objects are defined. Integer definitions are on the left, and equivalent character definitions are in corresponding positions on the right. Each global object is first declared to the assembler as an entry point. This is done with the directive:

```
PUBLIC name
```

As you can see, the compiler prefixes each name with a single underscore character, which serves to avoid clashes with assembler reserved words.

DW (define word) directives define integers, and **DB** (define byte) directives do the same for characters. If no initial value is specified,

```
n DUP (0)
```

is used to allocate ***n*** occurrences of the value zero. (Recall that uninitialized globals are guaranteed to start with initial values of zero.) However, if initial values are given, then the individual values are listed, each resulting in an occurrence of an object. Notice that when fewer initial values than objects are given, the ***n* DUP (0)** is used to define the uninitialized trailing objects.

Note: The names of the objects in these examples are abbreviations of each object type. For example, **gi** is a *global integer;* **gca** is a *global character* array; and **gip** is a *global integer pointer.* The examples which follow use the same naming conventions, except that **l** means *local,* **a** means *argument,* and **e** means *external.*

Input	Output	Input	Output
`int`		`char`	
`gi,`		`gc,`	
	`PUBLIC _GI`		`PUBLIC _GC`
	`_GI DW 1 DUP(0)`		`_GC DB 1 DUP(0)`
`gi2 = 123,`		`gc2 = 'a',`	
	`PUBLIC _GI2`		`PUBLIC _GC2`
	`_GI2 DW 123`		`_GC2 DB 97`
`gia[10] = {1, 2, 3},`		`gca[10] = "abc",`	
	`PUBLIC _GIA`		`PUBLIC _GCA`
	`_GIA DW 1,2,3`		`_GCA DB 97,98,99,0`
	`DW 7 DUP(0)`		`DB 6 DUP(0)`
`*gip;`		`*gcp;`	
	`PUBLIC _GIP`		`PUBLIC _GCP`
	`_GIP DW 0`		`_GCP DW 0`

Listing 19-2. Code Generated by Global Objects

Global References

The examples in Listing 19-3 illustrate the code generated when global objects are referenced. Again, integer and character examples are separated into left- and right-hand columns. Each example is written as a very simple expression statement—just the reference in question. This isolates the references for the sake of illustration. Of course, the same code is generated when these references occur in more complicated expressions.

Compare the references to integers and characters. In the first case, **gi** is obtained by moving a word from it's place in memory as indicated by the label **_GI.** The assembler knows to move a word instead of a byte because the destination is a 16-bit register. In contrast, the reference to **gc** moves only a byte to AL. It also executes a **CBW** to convert from a byte in AL to a word in AX by means of sign extension.

Next, look at the effect of placing an address operator (**&**) in front of these references. Since this calls for the address of the object, the **OFFSET** operator is given to the assembler so that the value of the label itself is moved to AX, rather than the object at that address. Notice here that there is no difference between the code for an integer address and a character address.

Following these are references to array names, but without subscripts. This should yield the address of the array. Not surprisingly, therefore, it generates the same code as the address operator applied to a variable.

When a subscript of value zero is used, the compiler simply skips the subscript arithmetic altogether and fetches the object at the array address. For example, **gia[0]** obtains the first integer in **gia,** just as **gi** obtains the integer **gi.** Now, notice that the indirection operator (*) applied to an array name has the same effect as a zero subscript—it obtains the *object* at the designated address.

The next two examples in Listing 19-3 illustrate ordinary array subscripting. **Gia[5]** refers to the sixth integer in **gia** by specifying the source operand as **_GIA+10.** The assembler evaluates this expression by adding ten to the address of **gia** to determine the location of the desired integer. Since this is done at assembly time rather than run time, it has no effect on program performance.

Input	Output	Input	Output
gi;	MOV AX,_GI	gc;	MOV AL,_GC CBW
&gi;	MOV AX,OFFSET _GI	&gc;	MOV AX,OFFSET _GC
gia;	MOV AX,OFFSET _GIA	gca;	MOV AX,OFFSET _GCA
gia[0];	MOV AX,_GIA	gca[0];	MOV AL,_GCA CBW
*gia;	MOV AX,_GIA	*gca;	MOV AL,_GCA CBW
gia[5];	MOV AX,_GIA+10	gca[5];	MOV AL,_GCA+5 CBW
*(gia + 5);	MOV AX,_GIA+10	*(gca + 5);	MOV AL,_GCA+5 CBW
gip;	MOV AX,_GIP	gcp;	MOV AX,_GCP
*gip;	MOV BX,_GIP MOV AX,[BX]	*gcp;	MOV BX,_GCP MOV AL,[BX] CBW
gip[5];	MOV BX,_GIP MOV AX,10[BX]	gcp[5];	MOV BX,_GCP MOV AL,5[BX] CBW
*(gip + 5);	MOV BX,_GIP MOV AX,10[BX]	*(gcp + 5);	MOV BX,_GCP MOV AL,5[BX] CBW

Listing 19-3. Code Generated by Global References

This is possible only because the subscript is a constant. If the subscript is a variable or a more complex expression, then more code is generated that must be

evaluated at run time. Comparing the integer to the character reference, we see that the subscript value of the integer reference is doubled, since integers occupy two bytes each. Also, a **CBW** instruction promotes the character to an integer.

The next examples illustrate the equivalence of subscripting and writing the address arithmetic directly. Notice that the same code is generated by **gia[5];** and ***(gia+5);.**

The next examples in Listing 19-3 serve to compare pointer references to the previous array references. At the source level, these are conceptually the same—unadorned references yield an address, both may be subscripted, and both may have address arithmetic performed on them. They differ fundamentally, however, in that an array name is not an lvalue, since it represents the constant address, whereas a pointer name is an lvalue, since it identifies a piece of memory which can be changed. Thus, an unadorned pointer name like **gip** produces an address by fetching the word at **_GIP** instead of its address. Since a Small C pointer is always two bytes long, regardless of whether it refers to characters or integers, the integer pointer reference generates the same code as the character pointer reference. They both fetch the contents of the pointer which is assumed to be an address. Of course, it is the programmer's responsibility to see that the pointer does, in fact, contain the correct address value.

Placing an indirection operator (*) before a pointer fetches the object pointed to. Thus, ***gip** first fetches the contents of the pointer into the secondary register BX, from which it can serve as a base address. Then, by means of:

```
MOV AX,[BX]
```

it moves the word pointed to by BX into AX. The brackets can be read as *contents of the memory location*. This must be a two-step operation, because the value of the pointer is a variable. It may have a different value with each execution of the reference. Comparing the integer and character examples, we see that, whereas each loads the pointer value the same way, the character is fetched into AL and then promoted to an integer.

Finally, the last examples in Listing 19-3 illustrate the code generated by subscripting pointers and performing address arithmetic on them. Note that these

are equivalent to array references except that the pointer's value provides the base address, whereas the array's name is itself a constant address.

External Declarations and References

The examples in Listing 19-4 illustrate the code generated when objects are declared external. In that case, there is no definition of the objects. They are simply declared external to the assembler by means of **EXTRN** directives.

Input	Output	Input	Output
`extern int`		`extern char`	
`ei,`	`EXTRN _EI:WORD`	`ec,`	`EXTRN _EC:BYTE`
`eia[10];`	`EXTRN _EIA:WORD`	`eca[10];`	`EXTRN _ECA:BYTE`

Listing 19-4. Code Generated by External Declarations

Listing 19-5 shows that references to external objects generate code which is identical to that which is generated for ordinary global objects. After all, they really are global objects; they just exist in another program module. It is up to the linker to determine the actual address for each of these references once it has concatenated the several parts of the program into a single code segment and a single data segment.

Input	Output	Input	Output
`ei;`	`MOV AX,_EI`	`ec;`	`MOV AL,_EC` `CBW`
`eia;`	`MOV AX,OFFSET _EIA`	`eca;`	`MOV AX,OFFSET _ECA`

`eia[5];`	`MOV AX,_EIA+10`	`eca[5];`	`MOV BX,OFFSET _ECA+5`
			`MOV AL,[BX]`
			`CBW`

Listing 19-5. Code Generated by External References

Local Declarations and References

The function in Listing 19-6 illustrates how local objects are both declared and referenced. The function proceeds from the top of the left column to the bottom of the right column. As before, integer references are on the left and equivalent character references are on the right.

First, notice that on entry to the function, the base pointer (BP) is saved on the stack and the new stack pointer value (SP) is moved to BP as the base of the stack frame for this function call. If arguments had been passed to this function, they would have been located at positive displacements from BP, beneath the saved value of BP and the return address. On the other hand, local objects are created on top of the stack; therefore, they are accessed by negative displacements from BP.

Rather than decrement SP separately for each local object, the compiler defers until the first executable statement is found. At that point, a single adjustment to SP allocates all locals for the current block. In this example, we define a character, a character array of 10 elements, and a character pointer—a total of 13 bytes. Then we define an integer, an integer array of 10 elements, and an integer pointer—a total of 24 bytes. These numbers combined account for the negative adjustment to SP in the instruction:

```
ADD SP,-37
```

In its symbol table, the compiler keeps track of the displacement from BP to each variable. Decrementing SP simply ensures that the stack space claimed by these locals will not be used for other purposes; it reserves the space. Notice that all local references involve a source operand of the form:

```
-n[BP]
```

where ***n*** is the displacement. When an address is needed, the *load effective address* instruction **LEA** is used. This instruction loads the address of the source operand rather than the operand itself.

Other than the fact that locals are located relative to **BP,** rather than by

means of labels, there is no difference between local references and global references. This can be verified by comparing listings 19-3 and 19-6.

Input	Output	Input	Output

`func () {`	`PUBLIC _FUNC`
	`_FUNC:`
	`PUSH BP`
	`MOV BP,SP`		

`char lc,`
`int`
	`lca[10], *lcp;`
	`li, lia[10], *lip;`
	`ADD SP,-37`		

`li;`	`MOV AX,-15[BP]`	`lc;`	`MOV AL,-1[BP]` `CBW`
`lia;`	`LEA AX,-35[BP]`	`lca;`	`LEA AX,-11[BP]`
`lia[0];`	`MOV AX,-35[BP]`	`lca[0];`	`MOV AL,-11[BP]` `CBW`
`*lia;`	`MOV AX,-35[BP]`	`*lca;`	`MOV AL,-11[BP]` `CBW`
`lia[5];`	`MOV AX,-25[BP]`	`lca[5];`	`MOV AL,-6[BP]` `CBW`
`*(lia + 5);`	`MOV AX,-25[BP]`	`*(lca + 5);`	`MOV AL,-6[BP]` `CBW`
`lip;`	`MOV AX,-37[BP]`	`lcp;`	`MOV AX,-13[BP]`

`lip[0];`	`MOV BX,-37[BP]`
	`MOV AX,[BX]`
		`lcp[0];`	`MOV BX,-13[BP]`
			`MOV AL,[BX]`
			`CBW`

Listing 19-6. Code Generated by Local Objects/References

Input	Output	Input	Output

`*lip;`	`MOV BX,-37[BP]`
	`MOV AX,[BX]`
		`*lcp;`	`MOV BX,-13[BP]`
			`MOV AL,[BX]`
			`CBW`

`lip[5];`	`MOV BX,-37[BP]`
	`ADD BX,10`
	`MOV AX,[BX]`
		`lcp[5];`	`MOV BX,-13[BP]`
			`ADD BX,5`
			`MOV AL,[BX]`
			`CBW`

`*(lip + 5);`	`MOV BX,-37[BP]`
	`ADD BX,10`
	`MOV AX,[BX]`
		`*(lcp + 5);`	`MOV BX,-13[BP]`
			`ADD BX,5`
			`MOV AL,[BX]`
			`CBW`

		`}`	`MOV SP,BP`
			`POP BP`
			`RET`

Listing 19-6 continued. Code Generated by Local Objects/References

Function Declarations and Calls

The example in Listing 19-7 shows the code generated when a function with arguments is declared and when the arguments passed to it are referenced. Again, integer references are on the left and character references are on the right.

In this example, observe that while arguments are referenced like locals, there are some differences. First, and most importantly, there is no allocation of arguments in the called function. Instead, the arguments are pushed onto the stack at the point of the function call.

The function assumes that the arguments are already on the stack when it receives control. If the wrong number or type of arguments are passed, the function forges ahead anyway, just as if all of the required arguments were present.

Another difference is that arguments are on the stack *beneath* the base address in BP rather than above it. As explained above, BP points to its original (saved) value on the stack. Beneath that is the return address which was placed

on the stack by the **call** instruction, followed by the arguments. Since the arguments are pushed onto the stack in the order of their appearance in the function call, the last argument is found, immediately below the return address, at **BP+4.** The next to last argument is at **BP+6,** and so on. You should recall that arrays cannot be passed. When an array name is given as an argument, the array's address is passed. Furthermore, since such an address exists in memory and can be changed, it is actually a pointer. This fact can be used to advantage within the function.

The last statement of the function is:

```
return (gi1);
```

which fetches the global integer **gi1** into AX as the return value, and generates the return sequence:

```
POP BP
RET
```

If local variables had been declared, the **pop** would have been preceded by an adjustment to SP to deallocate the locals and return SP to the saved value of BP.

If the **return** statement is removed, the return sequence would be generated automatically. In that case, however, no return value would be established. Whatever happens to be in AX (actually ***acp**) will be the return value.

Input	Output	Input	Output
`func(ai, aia, aip, ac, aca, acp)`		`int ai, aia[], *aip;` `char ac, aca[], *acp; {`	
	`PUBLIC _FUNC`		
`_FUNC:`			
	`PUSH BP`		
	`MOV BP,SP`		
`ai;`	`MOV AX,14[BP]`	`ac;`	`MOV AL,8[BP]` `CBW`
`aia;`	`MOV AX,12[BP]`	`aca;`	`MOV AX,6[BP]`

`aia[5];`	`MOV BX,12[BP]`
	`ADD BX,10`
	`MOV AX,[BX]`
		`aca[5];`	`MOV BX,6[BP]`
			`ADD BX,5`
			`MOV AL,[BX]`
			`CBW`

`aip;`	`MOV AX,10[BP]`	`acp;`	`MOV AX,4[BP]`

`*aip;`	`MOV BX,10[BP]`
	`MOV AX,[BX]`
		`*acp;`	`MOV BX,4[BP]`
			`MOV AL,[BX]`
			`CBW`

		`return (gi1);`	

			`MOV AX,_GI1`
			`POP BP`
			`RET`

		`}`	

Listing 19-7. Code Generated by Function Arguments/References

The example in Listing 19-8 shows that the code generated when a function is called is quite straightforward. First, each argument expression is evaluated and pushed onto the stack. The optimizer sees to it that the global lvalues are pushed directly from memory onto the stack. Then a count of the number of arguments is loaded into the CL register. Next, a call to the function is per-

formed. On return from the function, the arguments are deallocated from the stack by adding 6 to the stack pointer.

Input	Output

`func(gi, gia, gip);`	`PUSH _GI`
	`MOV AX,OFFSET _GIA`
	`PUSH AX`
	`PUSH _GIP`
	`MOV CL,3`
	`CALL _FUNC`
	`ADD SP,6`

Listing 19-8. Code Generated by Direct Function Calls

Another type of function call is required when the function address is calculated. The example in Listing 19-9 illustrates this situation. Notice here that the function address (contained in **gia[5]**) is pushed onto the stack. Then, as each argument is evaluated, the sequence:

```
POP     BX
XCHG    AX,BX
PUSH    BX
```

swaps the argument in AX with the function address on the stack. This leaves the function address in AX. If another argument follows, however, the function address is pushed onto the stack again. That way it floats on top of the stack as arguments are processed. Then, when all of the arguments have been processed, the address is not pushed onto the stack, but remains in AX. As usual, a count of the number of arguments is loaded into CL. Finally, the function is called with:

```
CALL AX
```

which calls the function pointed to by AX. On return, the arguments are deallocated by incrementing SP.

Input	Output

`gia[5] (gi, gc);`	`MOV AX,_GIA+10`
	
	`PUSH AX`
	`MOV AX,_GI`
	`POP BX`
	`XCHG AX,BX`
	`PUSH BX`
	
	`PUSH AX`
	`MOV AL,_GC`
	`CBW`
	`POP BX`
	`XCHG AX,BX`
	`PUSH BX`
	
	`MOV CL,2`
	`CALL AX`
	`ADD SP,4`

Listing 19-9. Code Generated by Indirect Function Calls

Expressions

It would be easy to go on with examples of interesting expressions. For the sake of brevity, however, only a representative sample of the various operators, and one example of a fairly complex expression are illustrated. Also for simplicity, only global objects are referenced. You can infer from the previous examples the effects of referencing locals and arguments.

The examples in Listings 19-10 and 19-11 show the effects of unary operators. The logical NOT operator is implemented as a call to **__lneg** in the **CALL** module. This routine logically negates the value in AX and leaves the result in AX.

Input	Output
`!gi;`	`MOV AX,_GI` `CALL __LNEG`

Listing 19-10. Code Generated by the Logical NOT Operator

The increment operator (Listing 19-11), applied as a prefix, adds 1 to the global integer **gi** in memory, and leaves the incremented value in AX as the value of the operation.

Input	Output

`++gi;`	`MOV AX,_GI`
	`INC AX`
	`MOV _GI,AX`

Listing 19-11. Code Generated by the Increment Prefix

Observe the difference in Listing 19-12 after two changes are made. By referring to an integer pointer rather than an integer, the increment value becomes 2 so that the pointer will advance to the next integer. Then by applying the increment operator as a suffix, the original value of the operand remains in AX. The target for the add operation is the pointer itself in memory.

Input	Output
`gip++;`	`MOV AX,_GIP` `ADD _GIP,2`

Listing 19-12. Code Generated by the Increment Suffix

In Listing 19-13 the indirection operator is applied once, twice, and three times to the integer pointer **gip.** In the first case, the word pointed to by **gip** is fetched by the sequence:

```
MOV BX,_GIP
MOV AX,[BX]
```

in which the first move obtains the pointer value in BX, and the second move uses it as the address from which the object is moved to AX. In the second case, that object is itself used as the address of the sought object. Therefore, the instructions:

```
MOV BX,AX
MOV AX,[BX]
```

are added to the sequence. These move the first object to BX from which it points to the final object which is loaded into AX.

The third case illustrates that there is no limit to the levels of indirection that can be applied.

Input	Output
`*gip;`	`MOV BX,_GIP` `MOV AX,[BX]`

`**gip;`	`MOV BX,_GIP`
	`MOV AX,[BX]`
	`MOV BX,AX`
	`MOV AX,[BX]`

`***gip;`	`MOV BX,_GIP`
	`MOV AX,[BX]`
	`MOV BX,AX`
	`MOV AX,[BX]`
	`MOV BX,AX`
	`MOV AX,[BX]`

Listing 19-13. Code Generated by the Indirection Operator

In Listing 19-14 the address operator is applied to a global pointer and a global array element. In the first case, the address of **gip** is loaded into AX. Notice that this obtains the address of the pointer, not the address it contains.

Applied to the array element **gia[5],** the address operator first loads the address of the array, then offsets it to the sixth element. This could be optimized to:

```
MOV AX,OFFSET _GIA+10
```

in which the addition is performed at assembly time rather than run time. This is left as an exercise.

Input	Output
`&gip;`	`MOV AX,OFFSET _GIP`
`&gia[5];`	`MOV AX,OFFSET _GIA` `ADD AX,10`

Listing 19-14. Code Generated by the Address Operator

The division and modulo operators (Listing 19-15) generate identical code except for the **MOV AX,DX** generated by the modulo operator. Since **IDIV** returns the quotient in AX and the remainder in DX, this move substitutes the remainder for the quotient as the value of the operation.

Input	Output

`gi / 5;`	`MOV BX,_GI`
	`MOV AX,5`
	`XCHG AX,BX`
	`CWD`
	`IDIV BX`

`gi % 5;`	`MOV BX,_GI`
	`MOV AX,5`
	`XCHG AX,BX`
	`CWD`
	`IDIV BX`
	`MOV AX,DX`

Listing 19-15. Code Generated by Division and Modulo Operators

The addition operator (Listing 19-16) is not particularly remarkable. It simply evaluates the left and right operands and performs the addition. Since the right operand is so simple, the optimizer has combined its evaluation with the add operation.

Input	Output
`gi + 5;`	`MOV AX,_GI` `ADD AX,5`

Listing 19-16. Code Generated by the Addition Operator

The relational operators, illustrated by the equality operator in Listing 19-17, are not much different from the other binary operators. A library routine, in this case **__eq,** is called to perform the comparison. If the stated condition is true, one is returned in AX; otherwise, zero is returned. Notice, here, that the left operand is evaluated directly in BX, rather than in AX then moved to BX. Here again, the optimizer has been at work.

Input	Output

`gi == 5;`	`MOV BX,_GI`
	`MOV AX,5`
	`CALL __EQ`

Listing 19-17. Code Generated by the Equality Operator

The logical AND (Listing 19-18) is interesting because during the process of evaluating a series of them, the first instance to yield *false* terminates the process. The label **_8** is the *false* exit point. Any subexpression yielding *false* causes a jump to that label where zero (*false*) is loaded into AX and control proceeds on to whatever follows.

Note: Since the three subexpressions are tested against zero (*false*), in-line code is generated rather than a call to one of the comparison routines in the **CALL** module. In each case, a bitwise OR of AX is performed on itself; this leaves the register unchanged, but sets the CPU's flags. The zero flag is tested by

JNE $+5 which jumps around the following **JMP _8** if the flag is not set (AX is not zero). The expression **$+5** tells the assembler to jump five bytes ahead of the current location; and, since the **jump** instructions occupy two and three bytes respectively, this targets the next instruction.

It may seem wasteful to use two **jump** instructions back to back when a simple:

```
JE _8
```

would do the job. However, the 8086 CPU does not support conditional jumps to arbitrary addresses. Its conditional jumps are relative to the instruction pointer (IP) and only an 8-bit signed displacement is used. This limits the range to -128 or +127 bytes. Since the subexpressions could be any size, we must use an unconditional jump (which has no such limit) to reach **_8.**

(Notice that the instruction at **_8** is redundant. Chapter 28 suggests improvements to this code.)

The logical OR operator is exactly opposite; the first term to yield true terminates the process with a value of 1 (*true*).

Notice that the subexpressions operated on do not have to contain relational operators. The last term **gc,** for instance, yields its own value, which is taken for *true* if it is not zero, and *false* otherwise.

Input	Output

`gi > 1 && gi < 5 && gc;`	`MOV BX,_GI`
	`MOV AX,1`
	`CALL __GT`
	`OR AX,AX`
	`JNE $+5`
	`JMP _8`
	
	`MOV BX,_GI`
	`MOV AX,5`
	`CALL __LT`
	`OR AX,AX`
	`JNE $+5`
	`JMP _8`
	
	`MOV AL,_GC`
	`CBW`
	`OR AX,AX`
	`JNE $+5`
	`JMP _8`
	`MOV AX,1`
	`JMP _9`
	
	`_8:`
	`XOR AX,AX`
	`_9:`

Listing 19-18. Code Generated by the Logical AND Operator

The assignment operators in Listing 19-19 are typical of all assignment operators. First, observe the simple assignment. Again the optimizer has been at work to produce a single instruction for setting **gi** to 5.

The += assignment is a bit more involved. Notice that it has the same effect as the expression:

```
gi = gi + 5
```

Gi is fetched, 5 is added to it, and the result is moved back to **gi.** Since the result remains in AX, it also becomes the value of the += operator.

Input	Output
`gi = 5;`	`MOV _GI,5`

`gi += 5;`	`MOV AX,_GI`
	`ADD AX,5`
	`MOV _GI,AX`

Listing 19-19. Code Generated by Assignment Operators

The last example of an expression (Listing 19-20) ties together several of the concepts presented above and also illustrates the flexibility with which all types of operators can be combined. This expression first calls **func()** and assigns its returned value to **gc.**

It then compares that value to 5, yielding either *true* (1) or *false* (zero). This in turn multiplies **'y'** (ASCII value 121) for the value to assign to **gi.** So **gi** is set to 121 if the function returns 5, otherwise zero. In either case, **gc** is set to the value returned by **func().**

Input	Output

`gi = 'y' * ((gc = func()) == 5);`	`XOR CL,CL`
	`CALL _FUNC`
	`MOV _GC,AL`
	`MOV BX,AX`
	`MOV AX,5`
	`CALL __EQ`
	`MOV BX,121`
	`IMUL BX`
	`MOV _GI,AX`

Listing 19-20. Code Generated by a Complex Expression

Statements

Following are samples of the code generated for each of the statements known to the compiler. First, a simple **if** statement (Listing 19-21) is presented. In this example, the expression being tested is only a variable. It is loaded into AX where it is tested for *true* or *false*. The same in-line logic as we saw in Listing 19-18 is used. Only if **gi** is not zero is 5 assigned to it.

Input	Output

`if (gi) gi = 5;`	`MOV AX,_GI`
	`OR AX,AX`
	`JNZ $+5`
	`JMP _10`
	`MOV _GI,5`
	`_10:`

Listing 19-21. Code Generated by an IF Statement

The **if** statement in Listing 19-22 contains an **else** clause. The first controlled statement:

```
gi = 5;
```

is executed if **gi** yields *true* and the second statement:

```
gi = 10;
```

is executed if it yields *false*. One and only one of these statements is executed. Notice that the first controlled statement is terminated by a **jump** around the second one.

Input	Output

`if (gi) gi = 5;`	`MOV AX,_GI`
	`OR AX,AX`
	`JNZ $+5`
	`JMP _11`
	`MOV _GI,5`
	`JMP _12`

`else gi = 10;`	`_11:`
	`MOV _GI,10`
	`_12:`

Listing 19-22. Code Generated by an IF/ELSE Statement

The two **if** statements in Listing 19-23 illustrate the greater efficiency of the code generated when a test is made against zero rather than non-zero values. The first case involves loading the constant 1 into AX and then testing it by calling **__ne** in the **CALL** module. This takes more space and time than the second example in which no value has to be loaded and no routine called.

Input	Output

`if (gi != 1) gi = 5;`	`MOV BX,_GI`
	`MOV AX,1`
	`CALL __NE`
	`OR AX,AX`
	`JNZ $+5`
	`JMP _13`
	`MOV _GI,5`
	`_13:`

`if (gi != 0) gi = 5;`	`MOV AX,_GI`
	`OR AX,AX`
	`JNE $+5`
	`JMP _14`
	`MOV _GI,5`
	`_14:`

Listing 19-23. Code Generated by Non-zero and Zero Tests

The **switch** statement is the most complicated of the Small C statements, but it is not really difficult to grasp. A typical **switch** statement is shown in Listing 19-24. In this example, notice that the immediate effect of a **case** or **default** prefix is to generate a label.

Later, at the end of the statement, a pair of words is generated for each **case.** They consist of the address (label) of the **case** and the value of the **switch** expression which targets that address.

First, the **switch** expression **gc** is evaluated, and placed in AX. Then a **jump** around the statements controlled by the **switch** is performed. Here a call is made to **__switch** which scans the following list of address/value word pairs looking for a match with the value in AX.

On the first match, **__switch** jumps to the corresponding label. If no match is found (detected by the zero at the end of the list), control goes to the point following the list. The **jump** found here is created by the **default** prefix. If there is no **default** there is no **jump,** and control simply skips over the statements controlled by the **switch** statement.

Notice that consecutive **case** prefixes work fine and provide a means for targeting several values of the **switch** expression to the same point. This example makes it clear that once control begins at some point within the statements controlled by the **switch** statement, it falls straight down through the remaining statements without regard for subsequent **case/default** prefixes.

To break the fall, a **break, continue,** or **goto** statement is required. Notice that the **break** statement generates a **jump** to the terminal label. Also notice that a similar **jump** follows the last controlled statement. This prevents control from reaching the call to **__switch** again.

Input	Output

`switch (gc) {`	`MOV AL,_GC`
	`CBW`
	`JMP _17`

`case 'Y':`	`_18:`

`case 'y': gcp = "yes";`	`_19:`
	`MOV AX,OFFSET _7+0`
	`MOV _GCP,AX`

`break;`	`JMP _16`
`case 'N':`	`_20:`

`case 'n': gcp = "no";`	`_21:`
	`MOV AX,OFFSET _7+4`
	`MOV _GCP,AX`

`break;`	`JMP _16`

`default: gcp = "error";`	`_22:`
	`MOV AX,OFFSET _7+7`
	`MOV _GCP,AX`

`}`	`JMP _16`

	`_17:`
	`CALL __SWITCH`
	`DW _18`
	`DW 89`
	`DW _19`
	`DW 121`
	`DW _20`
	`DW 78`
	`DW _21`
	`DW 110`
	`DW 0`
	`JMP _22`

	`_16:`

Listing 19-24. Code Generated by a SWITCH Statement

The **while** statement has a particularly simple structure in assembly language, as the example in Listing 19-25 shows. In this example, the control variable **gi** is decremented with each iteration and checked for *true* or *false*. If it is *false* (zero) then a **jump** to the terminal label is performed. Otherwise, the function call is performed and a **jump** back to the top is executed. The loop continues until the tested expression becomes *false*.

Input	Output

```
while (-gi) func(gi);    _23:
                                MOV AX,_GI
                                DEC AX
                                MOV _GI,AX
                                OR AX,AX
                                JNZ $+5
                                JMP _24

                                PUSH _GI
                                MOV CL,1
                                CALL _FUNC
                                ADD SP,2
                                JMP _23
                         _24:
```

Listing 19-25. Code Generated by a WHILE Statement

The **for** statement (Listing 19-26) begins by evaluating the first of three expressions in parentheses. This is the initializing expression. Next, the second expression, the controlling expression, is evaluated and tested. If it yields *false*, a **jump** to the terminal label is performed. Otherwise, there is a jump to the controlled statement. After that, there is a jump back to the instructions which evaluate the third expression. This is where a control variable is usually incremented or decremented. After that, there is a jump back for another evaluation of the control expression, and the process repeats itself until the control expression evaluates *false*.

Input	Output

```
for (gi = 4; gi >= 0; -gi) gia[gi] = 0;
```

```
        MOV _GI,4
_27:
        MOV AX,_GI
        OR AX,AX
        JGE $+5
        JMP _26
        JMP _28
_25:
        MOV AX,_GI
        DEC AX
        MOV _GI,AX
        JMP _27
_28:
        MOV AX,_GI
        MOV BX,OFFSET _GIA
        SHL AX,1
        ADD BX,AX
        XOR AX,AX
        MOV [BX],AX
        JMP _25
_26:
```

Listing 19-26. Code Generated by a FOR Statement

The sample **for** statement in Listing 19-27 is interesting because it illustrates what happens when the three expressions in parentheses are omitted. If any of these expressions is missing, there is simply no code generated in its place. Eliminating all three expressions makes it equivalent to:

```
while (1) ...
```

The code is less efficient, however, because of the extra jumping around.

Input	Output

`for (;;) {gia[gi] = 0; if (++gi > 5) break;}`	`_31:`
	`  JMP _32`
	`_29:`
	`  JMP _31`
	`_32:`
	`  MOV AX,_GI`
	`  MOV BX,OFFSET _GIA`
	`  SHL AX,1`
	`  ADD BX,AX`
	`  XOR AX,AX`
	`  MOV [BX],AX`
	
	`  MOV AX,_GI`
	`  INC AX`
	`  MOV _GI,AX`
	`  MOV BX,AX`
	`  MOV AX,5`
	`  CALL __GT`
	`  OR AX,AX`
	`  JNZ $+5`
	`  JMP _33`
	
	`  JMP _30`
	`_33:`
	`  JMP _29`
	`_30:`

Listing 19-27. Code Generated by a FOR Without Expressions

The **do/while** statement is simply an inverted **while** in which the controlling expression is evaluated last. There is always at least one execution of the controlled statement. The example in Listing 19-28 illustrates the code generated for such a statement.

Input	Output

```
do gia[gi] = 0; while (--gi);  _34:
                                       MOV AX,_GI
                                       MOV BX,OFFSET _GIA
                                       SHL AX,1
                                       ADD BX,AX
                                       XOR AX,AX
                                       MOV [BX],AX

                                       MOV AX,_GI
                                       DEC AX
                                       MOV _GI,AX
                                       OR AX,AX
                                       JNZ $+5
                                       JMP _35
                                       JMP _34
                               _35:
```

Listing 19-28. Code Generated by a DO/WHILE Statement

As an example of the **goto** statement, consider Listing 19-29. Here a label is written, followed by an assignment statement and then a **goto** that targets the label. The source label translates to a unique numeric label since the same source label can be used legally in different functions. If the source label were used directly, the assembler might flag **"duplicate label"** errors. The **goto** itself simply generates an unconditional jump to the target label.

Input	Output

```
abc:        _36:
gi = 5;             MOV _GI,5
goto abc;           JMP _36
```

Listing 19-29. Code Generated by a GOTO Statement

Conclusion

This finishes our overview of the code generated by the Small C compiler. Many other situations could have been presented, but the combinations are endless. These should suffice to illustrate the task performed by the compiler, and to make the compiler's output understandable.

Of course, you can test the compiler further yourself. The simplest way is to invoke it without command-line arguments. Then enter source statements from the keyboard and watch on the screen what the compiler generates. Whenever there is doubt as to what the compiler does with some particular statement, just ask it.

CHAPTER 20

Data Structures

This chapter describes the seven main data structures used by the Small C compiler. They are: the input line, the literal pool, the macro pool, the staging buffer, the switch table, the symbol table, and the while queue. Gaining a conceptual understanding of these will make the remainder of the compiler much easier to learn.

The term *structure* is used here in a generic sense, not with reference to the C language construct of the same name. Although that feature of Full C is a real convenience, it is hardly essential for working with data structures of any kind and, since it is not supported by Small C, the compiler is written without them.

Space for these structures is allocated dynamically in **main()** when the compiler begins execution. In each case, a pointer receives the address of the memory block that is allocated for the structure. Table 20-1 lists these pointer names together with their data types and the amount of memory allocated. The memory sizes are expressed as macro names which are defined in the file **CC.H.**

Pointer	Type	Size	Description
pline	char	LINESIZE	input line (parsing)
mline	char	LINESIZE	input line (macro)
litq	char	LITABSZ	literal queue
macn	char	MACNSIZE	macro name buffer
macq	char	MACQSIZE	macro string buffer
stage	int	STAGESIZE	staging buffer
swnext	int	SWTABSZ	switch queue (next entry)
symtab	char	SYMTBSZ	symbol table
wq	int	WQTABSZ	while queue

Table 20-1. Data Structure Pointers

The Input Line

There are two input line buffers in the compiler—**pline,** from which parsing is done; and **mline,** from which macro processing is done.

The preprocessor reads from **mline** while writing into **pline,** from which the parser operates. The function which reads source lines into the compiler **inline()** places its data wherever the pointer **line** points. Before a line is read into the compiler, **line** is set to **mline,** causing the raw source code to be placed there. The preprocessor then scans **mline** looking for matches with defined symbols. Tokens that do not match are copied directly into **pline.** Tokens that match have their replacement text copied into **pline** instead. When the end of the line is reached, **line** is reset to point to **pline,** from which parsing takes place.

Associated with **line** is another global pointer, **lptr,** which points to the *current character* in **line. Lptr** marches along the line as tokens are being recognized. Since the same scanning functions are shared by the preprocessor and the parser, they are directed by **lptr** so that they work from either line buffer. They reference **mline** during preprocessing and **pline** while parsing.

The Literal Pool

The literal pool is really just a character/integer buffer. It serves two purposes: it temporarily holds strings until they can be dumped at the end of the current function, and it temporarily holds initializers that are applied to global objects. The latter case does not require buffering, but it is more convenient to use the literal pool and its functions than to write special functions just for global initializers. Since function definitions are global, there is no conflict in these two uses of the literal pool; they occur at different times.

In the first case, **dofunction(),** at the beginning of a function, clears the literal pool and allocates a label number for it. At the end of the function, it dumps the literal pool; that is, it generates the label followed by enough DB instructions to define the function's strings. Each string within the function is referenced by an offset from the label. Since the literal pool is reset with each function, it only has to be large enough to hold the strings in a single function.

In the second case, the literal pool is used by **initials()** to initialize global

objects. For each object, it is first cleared. Any initializers following the object's name cause constants to be stored in the pool. Finally, at the end of the definition, the contents of the pool are output as **DB** or **DW** instructions that cause the assembler to place the constants in memory.

The literal pool is controlled by two global variables—**litq** which points to the beginning of the pool, and **litptr** which is not really a pointer but a subscript to the next available position in the pool (the byte following the last one used).

The function **stowlit()** puts things into the pool, and **dumplits()** dumps it by generating a sufficient number of **DB** or **DW** instructions to define the contents of the pool to the assembler. These are described in Chapter 21.

The Macro Pool

The macro pool is made up of two separate structures: the macro name table, and the macro text queue that holds substitution text. The name table has the format shown in Figure 20-1.

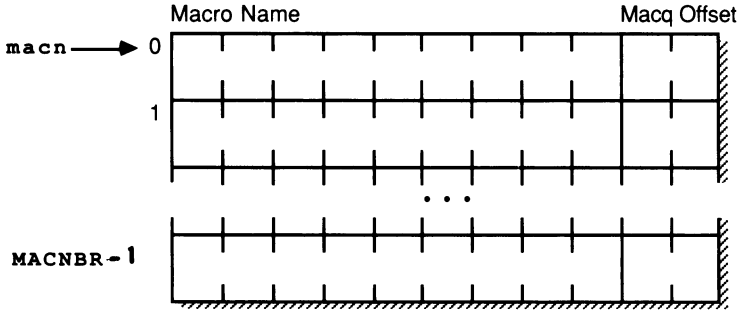


Figure 20-1. Format of the Macro Name Table

The name table consists of a fixed number of fixed-length entries. Each entry consists of a string for a macro name and an offset into **macq.** The names are placed in the name field left-justified with the customary trailing null byte; the offset is an ordinary integer which locates the beginning of the substitution text for the named macro.

Searches are performed on the name table by the same function that searches the global symbol table—**search()**. **Search()** employs a hashing algorithm to quickly locate a name in the table or the vacant entry where a name should be placed. On finding the name it returns *true*; otherwise, *false*. In either case, the global pointer **cptr** is set to the entry matching the name, or the vacant entry in which the name should be placed.

Macq is simply a character buffer into which the replacement text for macro definitions is stored. For each symbol, the entry in **macn** contains the offset to a null terminated string in **macq**.

The function **addmac()** processes **#define** statements by adding entries to **macn** and replacement text to **macq**. **Preprocess()** performs the substitutions.

The Staging Buffer

The staging buffer is a large integer buffer for holding generated code until **outcode()** can write it to the output file. It receives only code generated by expressions—the major part of C programs. Code that falls between expressions is not buffered in this way, but goes straight to the output file. The reason for buffering the code generated by expressions is so that it can be optimized. Some optimizing is done by the expression analyzer itself, but most is done by the peephole optimizer **peep()** as the buffered code is written to the output file.

Each entry in the staging buffer consists of two integers, a small numeric code called a *p-code* or *pseudo-code*, and an integer value that further qualifies the p-code. Each p-code represents one or more assembler instructions. A few represent partial instructions.

Two global pointers control the staging buffer. **Snext** either contains zero or points to the next available entry in the buffer, and **slast** points to the end of the buffer. When **snext** is zero, an expression is not being analyzed and so code generated by **gen()** goes directly to the output file. When **snext** is not zero, however, **gen()** places p-codes into the staging buffer at the position indicated.

Two functions, **setstage()** and **clearstage()**, manipulate **snext** and the contents of the buffer. **Setstage()** is called before expression analysis begins and before each subexpression is analyzed. In the first case, it changes **snext** from

zero to the starting address of the staging buffer (**stage**), thereby directing future output to the beginning of the buffer. In the second case, **snext** is left alone but saved so that code generated from that point can be ignored later by resetting **snext** to its original value. The expression analyzer does this when it sees that an expression or subexpression resulted in a constant value. In that case, it throws away the code that it generated and replaces it with a single instruction that loads the constant into the primary register.

Clearstage() either writes the contents of the buffer to the output file and resets **snext** to zero, or merely backs up **snext** to an earlier point in the buffer, thereby discarding the most recently generated code. **Clearstage()** calls **outcode()** to translate the p-codes to ASCII strings and write them to the output file. **Outcode()** in turn calls **peep()** to optimize the p-codes before it translates and writes them.

The Switch Table

For each **case** in a **switch** statement the **switch** table holds the value of the **case** and the number of the label to which control should go in that **case.** Figure 20-2 shows the format of the **switch** table.

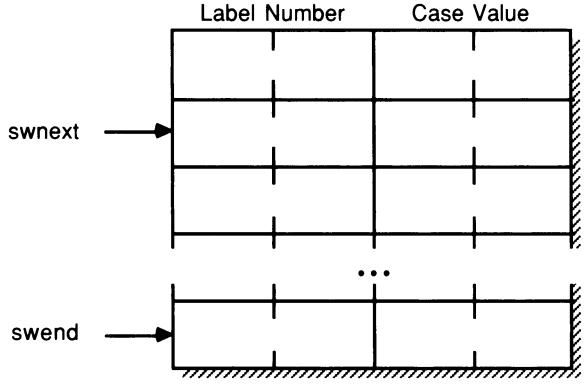


Figure 20-2: Format of the Switch Table

Each entry consists of two words: one for the label number and one for the value of the corresponding **case**. After compiling the body of the **switch** statement, which generates ordinary code interspersed with numeric labels for the cases, Small C generates a call to **__switch,** followed immediately by a list of label/value pairs to be scanned by **__switch** in deciding which label to target. The purpose of the switch table is to accumulate these label/value pairs until they can be dumped after the call to **__switch**.

Two global pointers control access to the table: **swnext** and **swend**. **Swnext** points to the next available entry in the table and **swend** always points to the last entry. Whenever a **case** is found in the body of a **switch**, a label is generated, its number is stored in the table, the value of the case is stored with it, and **swnext** is advanced to the next entry. An attempt to advance **swnext** beyond **swend** results in an error.

Notice that **switch** statements can be nested. Therefore, it is necessary to track label/value pairs for higher level statements as well as for the current one. This is done by partitioning the table at the start of every nested **switch**. The function **doswitch()** saves **swnext** in a local variable and then calls **statement()** recursively to compile the compound statement which is the body of the current **switch**. If another **switch** is encountered before **statement()** returns, **doswitch()** is again called (this time recursively), again saving **swnext** locally, and again calling **statement()**. When **statement()** does finally return, **doswitch()** generates its call to **__switch** and dumps the switch table from the entry indicated by **swnext** (when the current instance of **doswitch()** was called) to the last entry made. Finally, **swnext** is restored to its previous value and **doswitch()** exits.

The Symbol Table

The symbol table is the principle data structure of the compiler. It stores the names of every object declared in the program and everything the compiler must know about them. As Figure 20-3 shows, the symbol table is partitioned into two parts: a local table and a global table. Although they carry the same data, they are really distinct tables with different search algorithms.

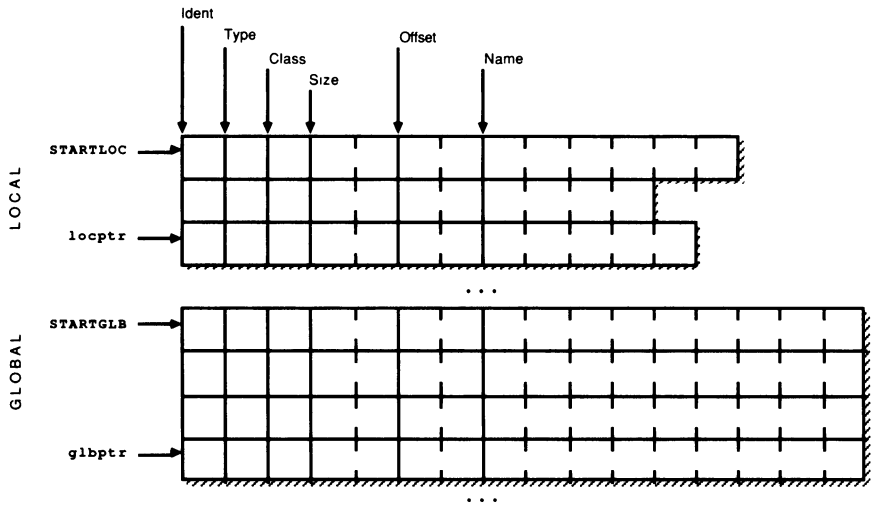


Figure 20-3. Format of the Symbol Table

A limited space at the front of the symbol table is reserved for local declarations. Its address is defined by the symbol **STARTLOC**. Since local declarations pertain to only one function at a time, not much space is needed. The symbol **NUMLOCS** determines how many local declarations will fit in the reserved space. The global pointer **locptr** controls placement of entries in the local table by pointing to the next available entry—the one immediately following the last one made.

The remainder of the table, beginning at the symbol **STARTGLB**, stores global declarations that, unlike local declarations, accumulate throughout the compilation run. The symbol **NUMGLBS** determines how many global declarations will fit in the space allocated. The global pointer **glbptr** is initially set to **STARTGLB** and thereafter always points to the start of the global table.

Each table entry consists of six fields—*identity*, *type*, *class*, *size*, *offset*, and *name*. The symbols **IDENT**, **TYPE**, **CLASS**, **SIZE**, **OFFSET**, and **NAME** specify the offsets, within an entry, to each of these fields.

The *identity* field tells what the declared entity is. The possible values are defined by the symbols shown in Table 20-2. Symbol table entries can represent

labels, variables, arrays of variables, pointers, and functions. These labels are the ones written in the source code, not the numeric labels created by the compiler. Labels exist only in the local symbol table.

Value	Symbol	Meaning
0	LABEL	declared label
1	VARIABLE	scalar variable
2	ARRAY	array of variables
3	POINTER	pointer
4	FUNCTION	function

Table 20-2. Defined Values for the Identity Field

The *type* field specifies the data type. The possible values are defined by the symbols shown in Table 20-3. Since data type is meaningless with respect to labels, this field does not apply to label entries, so zero is used. For variables, this field specifies the type of variable. For arrays, it tells the type of the array's elements. For pointers, it indicates the type of object referenced by the pointer. And for functions, it specifies the type of object returned—always integer. The values assigned to these symbols are especially chosen so that when they are shifted right by two bits, they yield the length of objects of the designated type.

Value	Symbol	Meaning
0	LABEL	not applicable
4	CHR	character data
8	INT	integer data
5	UCHR	unsigned character data
9	UINT	unsigned integer data

Table 20-3. Defined Values for the Type Field

The *class* field gives the storage class of an object. The possible values are defined by the symbols shown in Table 20-4. Since storage class is meaningless

with respect to labels, this field is set to zero for label entries. The idea of a storage class is the same for variables, arrays, and pointers. **STATIC** storage is always present. **STATIC** objects are declared to the assembler with labels and **DB** or **DW** directives. **AUTOMATIC** storage is dynamically allocated on the stack upon entry of a compound statement, and is deallocated on exit (recall that the body of a function is a compound statement). Automatic (local) objects are referenced by their offset from the stack frame pointer BP rather than by means of labels. **EXTERNAL** objects exist at the global level in some other, separately compiled, module. They are merely declared to the assembler to be external. **AUTOEXT** applies to functions which are referenced but not declared in the program. These are assumed to be external and are so declared to the assembler at the end of the compile run.

Value	Symbol	Meaning
0	LABEL	not applicable
1	AUTOMATIC	automatic storage
2	STATIC	static storage
3	EXTERNAL	declared external
4	AUTOEXT	assumed external

Table 20-4. Defined Values for the Class Field

The *size* field gives the number of bytes occupied by the object. If it is a scalar item, then the size is 1 for a character and 2 for an integer. The size of an array, however, is the total amount of space taken up by the array. For character arrays, this is the same as the number of elements, but for integer arrays it is twice that value.

The *offset* field specifies a numeric value (if applicable). This is primarily the stack frame offset for local objects. If the entry describes a label, this field contains the compiler-assigned label number to be used in place of the actual label name. Label names are not used directly because their scope is restricted to the functions in which they appear. The same label may occur many times in a program, so using the name directly could result in an error at assembly time.

Therefore, Small C associates, with each declared label, a unique compiler-generated label number which is used instead.

The *name* field contains the name of an entry as a character string. This is a variable-length field in the local table and a fixed-length field in the global table. In the local table, the name is terminated with a byte containing the length of the name (1–8). In the global table a null byte terminates the name and any remaining bytes are ignored.

The local table is searched sequentially backward, making it easy to implement the scope rules for local declarations. The inefficiency of this method is compensated for by the relatively small number of local declarations in any given function. Since local declarations can occur at the beginning of any compound statement and compound statements can be nested, searching backwards guarantees that when the same name occurs more than once the search will find the one declared at the lowest level. **Locptr** controls placement of entries in the local table. At the beginning of each function **dofunction()** resets it to **STARTLOC**, thereby emptying the table.

Also, at the beginning of each compound statement **compound()** saves **locptr** so it can restore it at the end, thereby causing the local table to forget declarations made in the block while remembering those made in superior blocks. The function **declloc()** is then used to process local declarations except labels. **Declloc()** in turn calls **addsym()** to actually place new symbols in the table; the same function adds entries to the global table also. The address of **locptr** is passed to **addsym()** so it will know where to place the new entry and so it can set **locptr** to the address of the next entry to be placed in the local table; i.e., the byte following the name in the current entry.

Since the name is terminated with a byte containing the length of the name, the local table can be searched backward even though it has variable length entries. The function **addlabel()** adds labels to the local table. The function **findloc()** searches the local table.

The global table is processed by means of a hashing algorithm. This compensates for the large number of entries which it usually contains. The function **findglb()** searches the global table by calling **search()**, the same function used

for macro name searches. Basically, **search()** hashes the sought symbol to pick an entry of choice. It then proceeds sequentially (and cyclically) from that point, looking for a match. A match or a vacant entry terminates the search. The terminating entry's address is placed in the global pointer **cptr**, and *true* is returned if there was a match, or *false* otherwise. If the search fails, then **cptr** points to the place where a new entry with the specified name should go.

The function **declglb()** processes global declarations. **Declglb()** in turn calls **addsym()**, the same function that adds entries to the local table, to place a new symbol in the global table. The address of **glbptr** is passed to **addsym(),** so it will know that it is working in the global table. In that case, it calls **findglb()** to locate a place for the new entry.

The While Queue

The *while queue* is used to store information needed for breaking out of and continuing **while**, **for**, and **do** loops, and for breaking out of **switch** statements. A queue is needed because these constructs can be nested. Obviously, the term *while queue* is a misnomer. At one time Small C supported only the **while** statement for loop control, and the **switch** statement was not supported. Then, with the addition of the new statements, the while queue was further employed but not renamed.

Figure 20-4 shows the format of the while queue. It is really just a short table of integers, three per entry. **WQTABSZ** specifies the length of the queue in integers, and so must be a multiple of three. **WQSIZ** defines the length of an entry (three) and **WQMAX** points to the last possible entry. The three items of information needed for continuing and breaking out of control constructs are: (1) the value of the compiler-relative stack pointer (**csp**) upon entry to the construct, (2) the number of the label that marks the continuation point of a loop, and (3) the number of the label that marks the exit point of the construct. The first item is needed so that the stack can be adjusted back to its original value, thereby deallocating any block-local variables that are no longer needed. The second and third are needed so that jumps can be generated which target the continuation and exit points, respectively.

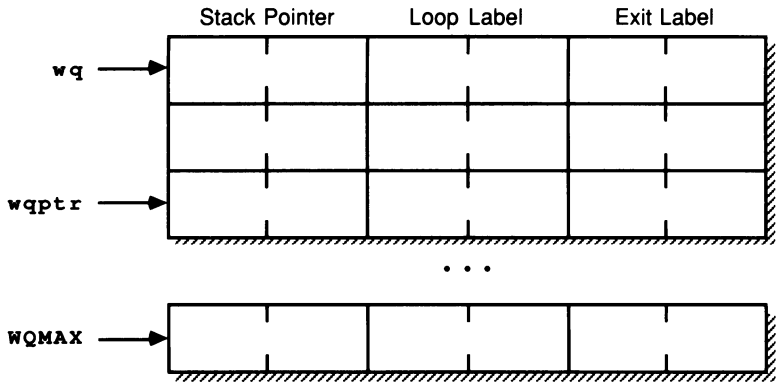


Figure 20-4. Format of the While Queue

The global pointer **wqptr** points to the next available entry in the queue. It is initially set to the address of the first entry, **wq**. As each **while**, **for**, **do**, or **switch** is encountered, an entry is made at **wqptr** which is then advanced to the next entry. This action is performed by **addwhile(),** which in turn calls **getlabel()** to create the two needed label numbers.

When a **break** or **continue** statement is found, **readwhile()** is called to fetch the address of the last entry in the queue. This is appropriate for **break** statements, but for **continue** statements, a superior entry might the desired one, since continues do not pertain to **switch** statements. If the last entries in the queue correspond to **switch** statements, then a backward search must be made for the entry representing the most nested loop statement—the last entry with a non-zero continue label number. This is done by repeatedly calling **readwhile()**, each time passing it the pointer it previously returned. With each call, **readwhile()** returns the next previous entry pointer.

Finally, at the end of the construct, **delwhile()** is called to remove the current (last) entry from the queue by backing up **wqptr** one entry. **Addwhile()** and **delwhile()** are called by **dowhile()**, **dofor()**, **dodo()**, and **doswitch()**. **Readwhile()** is called by **dobreak()** and **docont()**.

CHAPTER 21

Back End Functions

This chapter describes the back end of the Small C compiler. The functions that comprise this part all reside in the fourth source file—**CC4.C**. They fall into three general categories: (1) code generation functions, (2) code optimizing functions, and (3) output functions. Since optimizing is a major topic, and since the optimizing functions are pretty much self-contained, that part of the back end is covered in its own chapter (see Chapter 27).

As you reach each function's description in the following text, turn to that function in the listings (Appendix C) and familiarize yourself with it. Then continue studying it as you read the description. Refer to Appendix H to find the function in the listings.

Code Generation Functions

Header() **Header**() is called once from **main**() before parsing begins. It initiates the code and data segments each with a dummy word to ensure that all addresses are non-zero. The choice of a word, instead of a byte, preserves word alignment in the data segment. **Header**() also declares external each of the routines in the **CALL** module. This ensures that the **CALL** module will be linked with the program and that its routines can be accessed. Each segment is introduced by calling **toseg**().

Toseg() **Toseg**() is called whenever a transition between the code and data segments is needed. The global integer **oldseg** remembers which segment is current. It initially contains zero, meaning that neither segment is current. On entry, **toseg**() accepts an integer argument indicating which is to be the new segment. The value 1 (defined as DATASEG) indicates the data segment, 2 (defined as

CODESEG) indicates the code segment, and zero indicates that no segment is to be initiated, but the current segment is to be terminated.

First, if the new segment matches the current segment, no action is taken and control returns to the caller. That failing, a switch is to be made, so the current segment, if there is one, is terminated first. If **oldseg** equals **CODESEG**, then:

```
CODE ENDS
```

is generated, and if **oldseg** equals **DATASEG**, then:

```
DATA ENDS
```

is generated. If **oldseg** is zero, nothing is written.

Next, if **newseg** is non-zero, the new segment is initiated with either:

```
DATA SEGMENT PUBLIC
```

or,

```
CODE SEGMENT PUBLIC
ASSUME CS:CODE, SS:DATA, DS:DATA
```

depending on the value of **newseg**. Finally, **newseg** is moved to **oldseg** to remember which segment is current.

Trailer() **Trailer**() is called by **main**() after parsing is finished. It performs five tasks. First, it generates external references for all undeclared external functions. It does this by searching the global symbol table for function entries with a storage class of **AUTOEXT**—established by **primary**() in the expression analyzer. When it finds an undeclared name, **primary**() assumes it to be a function and establishes it as such in the global symbol table with this storage class. If the function should be defined later in the program, then the storage class would be changed to a different value so that it will not also be declared to the assembler as external.

Secondly, if the function **main**() exists in the global symbol table, then **_main**() is declared to the assembler to be external. This refers to a function in the library module **CSYSLIB** that sets up conditions for the program to initially receive control, and in turn calls **main**() within the program. This external declaration guarantees that **CSYSLIB** will be linked with the program. Subprograms that do not contain **main**() do not generate this external reference. This prevents a "**duplicate definition**" error at assembly time when **CSYSLIB.C** is compiled.

Next, whichever segment (data or code) is current at the end of the program is terminated by calling **toseg(),** with zero for the new segment. This terminates the current segment without opening another one.

Then, the assembler directive **END** is written to terminate the source file.

The last task performed by **trailer**() is to display optimizer statistics. This logic is not compiled into the production compiler. If the **#define** of **DISOPT** (in **CC4.C**) is "de-commented," however, the fourth part of the compiler is recompiled, and the compiler is relinked; then a count of the number of times each optimization was applied will be displayed at the end of each compile run. The display goes to the standard output file.

Public() **Public**() is called whenever a global object is being defined. It receives the identity code of the object so that it can distinguish functions from data. First, it calls **toseg**() to switch segments if necessary. Functions go into the code segment and global data goes into the data segment. Next, it writes:

```
PUBLIC name
```

(where ***name*** is the string in the global array **ssname**[] that contains the name being defined) to the output file. This tells the assembler that ***name*** is an entry point, allowing it to be referenced from separately compiled source files.

The name is then written at the beginning of a new line. If this call is for a data item, then the new line is left unfinished; later, a **DW** or **DB** directive will complete the line. If it is a function, however, then the name is terminated with a colon and a new line is started. This establishes the reference point for the start of the function.

External() **External**() is called whenever a name is to be declared external. It receives a pointer to the name, an integer giving the size of the object being declared, and an integer specifying the identity of the object. As with **public**(), it first establishes the correct segment. Then it writes:

```
EXTRN name:size
```

where ***name*** is the designated name in uppercase (with a leading underscore) and *size* is either **BYTE**, **WORD**, or **NEAR**. The first two of these apply to data declarations and the last to function declarations.

Outsize() **Outsize**() is called by **external**() to generate the *size* portion of its output line. It bases its decision on the object's size and identity, as indicated above.

Point() **Point**() simply writes:

```
DW $+2
```

to the output file. When this follows a label, it initializes a named pointer with the address of the following byte. This function is used to define pointers that are initialized with character strings. Since the string is stored immediately after the pointer, giving the pointer the value of the following byte effectively points it to the string.

Setstage() This function performs three tasks. First, it saves, in the pointer whose address is supplied by the caller, the current position in the staging buffer (**snext**). This is the original position, before **setstage**() changes it (if it does). Next, if **snext** is zero, meaning that the buffer is not being used, **setstage**() points it to the first word in the buffer, thus initializing the buffer for use and flagging the fact that the buffer is to be used thereafter until **clearstage**() resets **snext** to zero again. If **snext** is already non-zero, however, then the buffer is already being used, so **snext** is left unchanged. Finally, **setstage**() saves the new (possibly

changed) staging buffer position in another pointer whose address is also supplied by the caller.

To recap, this function initiates use of the staging buffer if it was not already being used. It also saves the previous and current buffer positions in pointers that the caller owns. These pointers will be the same unless the buffer was not previously in use (i.e., at the beginning of the analysis of an expression). In that case, the first will be zero and the latter will point to the beginning of the staging buffer.

Recall from Chapter 20 that the staging buffer is used only for holding code generated during expression evaluation. At other times it is bypassed; generated code is sent directly to the output file.

Gen() This is the primary code generation function. From the other function descriptions, we can see that some functions write directly to the output file; these are the exception rather than the rule. **Gen()** accepts two integer arguments—a *p-code* and a *value*. The value could be a signed integer, or the address of a symbol table entry. In many cases, depending on the p-code, the value serves no purpose at all. Whether or not the value is used, every p-code has one, and this function treats them all the same.

Gen() tests the p-code for one of several cases that receive special treatment. If the p-code is **GETb1pu**, **GETb1p**, or **GETw1p**, the compiler wants to generate code that fetches an operand pointed to by the primary register. But these codes expect to find the address in the secondary register. So **gen()** first calls itself recursively to generate **MOVE21,** which moves the address to the secondary register.

Likewise, p-codes **SUB12**, **MOD12**, **MOD12u**, **DIV12**, and **DIV12u** first generate **SWAP12** to swap the primary and secondary registers. This is because these operations are not commutative and the operands are in the wrong registers for the 8086 subtract and divide instructions.

Since pushes and pops must adjust the compiler-relative stack pointer **csp**, **PUSH1** and **POP2** are sensed and appropriate adjustments to **csp** are made.

Two additional p-codes, **ADDSP** and **RETURN**, receive special treatment.

These codes must adjust the machine stack pointer SP as well as the compiler-relative stack pointer. The values associated with these codes directly specify a new value for **csp**. First, that value is saved, then it is converted to an adjustment by subtracting **csp** from it; this adjustment will be added to SP to adjust the machine stack. Finally, **csp** is set to its new (saved) value.

At this point, **gen**() does its primary task. If **snext** is zero, the staging buffer is not being used, so **gen**() calls **outcode**() to translate the p-code and its value to an ASCII string in the output file. Control then returns to the caller.

If the staging buffer is in use, however, then an overflow test is performed. If successful, the p-code and its value are placed in the next two words of the buffer, **snext** is advanced by two, and control returns to the caller.

Clearstage() This function clears code from the staging buffer. It either discards part of the code that was placed into the staging buffer most recently, or it dumps the buffer to the output file and resets it to its inactive state (**snext** == 0). In the latter case, it may simply reset the buffer without writing to the output file.

Two pointer arguments are received—**before** points to the buffer position that preceded the start of the code being cleared, and **start** points to the start of the code being cleared. These are normally the values that were earlier saved by **setstage**(). However, **start** is sometimes forced to zero by the caller.

If **before** is not zero, the whole expression is not being cleared (only the most recently evaluated subexpression), so **snext** is set to that value, effectively ignoring code that was generated beyond that point. Control then returns to the caller. As we shall see in Chapter 26, the code generated by the analyzer while evaluating constant subexpressions is discarded and replaced by a single p-code that loads the result.

If **before** is zero, however, the entire expression is being cleared from the buffer (this call corresponds to the first **setstage**() call for the expression), so the staging buffer is dumped via **dumpstage**() to the output file and then deactivated. There are times when all of the code for an expression should be discarded. So, **dumpstage**() is called only if **start** is not zero. In either case, **snext** is reset to zero before returning to the caller. This deactivates buffering by causing **gen**() to write directly to the output file.

Dumpstage() As we saw above, this function is called from **clearstage**() to dump the staging buffer to the output file. It does this, one p-code at a time, by calling **outcode**(). If optimizing has not been disabled for the run, however, it first calls **peep**() for each optimization that might possibly be applied. If an optimization is applied, then the optimizing loop is restarted at the beginning so that the optimized p-code sequence can be further optimized. Only the current and subsequent p-codes come under the scrutiny of **peep**().

When **peep**() indicates that all attempts have failed, **dumpstage**() calls **outcode**() to translate and write the current p-code to the output file as an ASCII string. Having done that, it advances to the next p-code and repeats the process until the last p-code in the buffer has been processed.

On entry, **dumpstage**() sets the global pointer **stail** to the value of **snext,** which indicates the next unused position in the staging buffer. This marks the end of the data in the buffer. Then, **snext** is set back to the beginning of the buffer, from which it advances p-code by p-code as **dumpstage**() does its work.

The main loop, which steps from one p-code to the next, continues as long as **snext** is less than **stail**. The inner loop, in which **peep**() attempts to find an applicable optimization, continues until either an optimization is applied or all attempts fail. If **DISOPT** was defined when part 4 of the compiler was compiled, each successful optimization is displayed on the screen (**stderr**) as it occurs.

Dumplits() **Dumplits**() is called at the end of each function definition to dump the accumulated string constants from the literal pool to the output file. It also serves to dump initial values for global objects.

It receives an integer argument that specifies whether the pool contains byte- or word-sized values. To save space in the output file, as many as 10 values are listed with each **DB** or **DW** directive. Therefore, this function comprises two loops. The outer loop calls **gen**() to generate a **DB** or **DW** directive, depending on the value of the argument. Then the inner loop repeats for a maximum of 10 times, dumping successive values (bytes or words) from the literal pool as signed decimal strings. Commas separate the values. When 10 values have been dumped, a newline is written and the outer loop continues. If more values remain

to be dumped, another **DB** or **DW** is generated and the inner loop is reinitiated. When the literal pool has been exhausted, a newline terminates the current line and control returns to the caller.

Dumpzero() This is a simple function that dumps a specified number of zeroes (bytes or words) into the output file. It receives two arguments that specify the size of the items (byte or word) and the number to dump. It simply calls **gen()** to produce

```
DB n DUP(0)
```

or

```
DW n DUP(0)
```

where ***n*** is the number of items being dumped.

Output Functions

In this section, we look at each of the output functions except the one which translates p-codes to assembly language.

Colon() **Colon()** calls the library function **fputc()** to write a colon to the output file designated by the global integer **output.**

Newline() **Newline()** calls **fputc()** to write a newline character to the output file designated by the global integer **output**. Recall from Chapter 12 that the **put** library functions convert this character to two characters—a carriage return followed by a line feed. This is the standard end-of-line sequence in ASCII files. When sent to the screen or to a printer, it has the effect of locating the next character at the beginning of the following line.

Outdec() **Outdec()** accepts a signed integer that it writes to the output file as a signed decimal character string. It calls **fputc()** repeatedly as it writes to the output file designated by **output**.

Outname() **Outname**() accepts a pointer to a character string that it writes to the output file as a Small C name. It prefixes the string with an underscore character and translates the characters to uppercase as it repeatedly calls **fputc**() until the end of the string is reached. The output file is designated by the global file descriptor **output**. This function does not write a newline character.

Outstr() **Outstr**() accepts a pointer to a character string that it writes to the output file by repeated calls to **fputc**() until the end of the string is reached. This function does not write a newline character.

A call to **poll**() at the beginning permits the compiler to be interrupted while writing to the output file. It honors <control-S> pauses as well as a <control-C> termination of the compile run.

Outline() **Outline**() also accepts a pointer to a string that it writes to the output file. It differs from **outstr**() only in that it does append a newline to the end of the string. Thus, it writes an entire *line*, or perhaps the last part of a line. This function first calls **outstr**() to write the string, then **newline**() to terminate the line.

Small C P-codes

For reasons of efficiency, before being output, code is first generated in the form of *pseudo-codes* or *p-codes*. These are small integer values, each of which corresponds to some particular assembly language instruction, instruction sequence, or partial instruction. Each p-code consists of two parts, the *p-code* itself and an integer *value* which influences the form of the ASCII string that the p-code will become. The function **outcode**() is a specialized output function that translates a p-code and its value into ASCII assembly language.

Before looking into **outcode(),** however, we should first familiarize ourselves with the Small C p-codes. To improve readability, the p-codes are defined in **CC.H** as manifest constants. They have systematic names, so that even an unfamiliar p-code can usually be figured out simply by knowing the system.

Tables 21-2 and 21-3 list all of the Small C p-codes together with brief explanations of their effects. The p-codes in Table 21-2 are generated directly by the compiler, and those in Table 21-3 are generated by the optimizer as it operates on the compiler's p-codes.

Symbol	Meaning
0	the value zero
1	primary register (pr in comments)
2	secondary register (sr in comments)
b	byte
f	jump on false condition
l	current literal pool label number
m	memory reference by label
n	numeric constant
p	indirect reference through pointer in sr
r	repeated r times
s	stack frame reference
u	unsigned
w	word
_	incomplete instruction (sequence)

Table 21-1. Small C P-code Legend

Table 21-1 is a legend of the lowercase letters, digits, and special characters that are used in p-code names. In the explanations in Tables 21-2 and 21-3, the abbreviation *pr* refers to the primary register, and *sr* refers to the secondary register. Each p-code name is based on a verb that is written in uppercase letters. This verb indicates the basic operation performed at run time or assembly time by the p-code. Attached to this are numbers, lowercase letters, and/or special characters that further define the specific activity of the p-code. For example,

```
DIV12u
```

designates a divide operation. The registers involved are the primary register (1) and the secondary register (2). Whenever registers (1 or 2) are specified, the left-most (or only) register named receives the result of the operation. Thus, **DIV12u** yields the quotient in the primary register. Finally, the trailing letter **u** indicates that an unsigned operation is performed. Of course, a naming system like this cannot tell everything. In this case, nothing designates which register contains the divisor and which the dividend.

Table 21-2. Compiler-Generated P-codes

P-code	Effect
ADD12	add sr to pr
ADDSP	add to stack pointer
AND12	AND sr to pr
ANEG1	arithmetically negate pr
ARGCNTn	pass argument count to a function
ASL12	arithmetically shift left sr by pr into pr
ASR12	arithmetically shift right sr by pr into pr
CALL1	call function through pr
CALLm	call function directly
BYTE_	define bytes (part 1)
BYTEn	define byte of value n
BYTErO	define r bytes of value 0
COM1	one's complement pr
DBL1	double pr
DBL2	double sr
DIV12	divide pr by sr
DIV12u	divide pr by sr (unsigned)
ENTER	set stack frame upon function entry
EQ1Of	jump if (pr == 0) is false
EQ12	set pr TRUE if (sr == pr)

P-code	Effect
GE10f	jump if (pr >= 0) is false
GE12	set pr TRUE if (sr >= pr)
GE12u	set pr TRUE if (sr >= pr) (unsigned)
POINT11	point pr to function's literal pool
POINT1m	point pr to memory item through label
GETb1m	get byte into pr from memory through label
GETb1mu	get unsigned byte into pr from memory through label
GETb1p	get byte into pr from memory through sr ptr
GETb1pu	get unsigned byte into pr from memory through sr ptr
GETw1m	get word into pr from memory through label
GETw1n	get word of value n into pr
GETw1p	get word into pr from memory through sr ptr
GETw2n	get word of value n into sr
GT10f	jump if (pr > 0) is false
GT12	set pr TRUE if (sr > pr)
GT12u	set pr TRUE if (sr > pr) (unsigned)
WORD_	define word (part 1)
WORDn	define word of value n
WORDr0	define r words of value 0
JMPm	jump
LABm	define label m
LE10f	jump if (pr <= 0) is false
LE12	set pr TRUE if (sr <= pr)
LE12u	set pr TRUE if (sr <= pr) (unsigned)
LNEG1	logical negation of pr
LT10f	jump if (pr < 0) is false
LT12	set pr TRUE if (sr < pr)
LT12u	set pr TRUE if (sr < pr) (unsigned)
MOD12	modulo pr by sr
MOD12u	modulo pr by sr (unsigned)
MOVE21	move pr to sr

P-code	Effect
MUL12	multiply pr by sr
MUL12u	multiply pr by sr (unsigned)
NE10f	jump if (pr != 0) is false
NE12	set pr TRUE if (sr != pr)
NEARm	define near pointer through label
OR12	OR sr onto pr
POINT1s	point pr to stack item
POP2	pop stack into sr
PUSH1	push pr onto stack
PUTbm1	put pr byte in memory through label
PUTbp1	put pr byte in memory through sr ptr
PUTwm1	put pr word in memory through label
PUTwp1	put pr word in memory through sr ptr
rDEC1	decrement pr (may repeat)
REFm	finish instruction with label
RETURN	restore stack and return
rINC1	increment pr (may repeat)
SUB12	sub sr from pr
SWAP12	swap pr and sr
SWAP1s	swap pr and top of stack
SWITCH	call _SWITCH to find a switch's case
XOR12	XOR pr with sr

Table 21-2. Compiler-Generated P-codes

Table 21-3. Optimizer Generated P-codes

P-code	Effect
ADD1n	add n to pr
ADD21	add pr to sr
ADD2n	add immediate value n to sr
ADDbpn	add n to memory byte through sr pointer
ADDwpn	add n to memory word through sr pointer
ADDm_	add n to memory byte/word through label (part 1)
COMMAn	finish instruction with ",n"
DECbp	decrement memory byte through sr pointer
DECwp	decrement memory word through sr pointer
POINT2m	point sr to memory through label
POINT2m_	point sr to memory through label (part 1)
GETb1s	get byte into pr from stack
GETb1su	get unsigned byte into pr from stack
GETw1m_	get word into pr from memory through label (part 1)
GETw1s	get word into pr from stack
GETw2m	get word into sr from memory (label)
GETw2p	get word into sr through sr pointer
GETw2s	get word into sr from stack
INCbp	increment byte in memory through sr pointer
INCwp	increment word in memory through sr pointer
PLUSn	finish instruction with "+n"
POINT2s	point sr to stack
PUSH2	push sr
PUSHm	push word from memory through label
PUSHp	push word from memory through sr pointer
PUSHs	push word from stack
PUT_m_	put byte/word into memory through label (part 1)
rDEC2	decrement sr (may repeat)

P-code	Effect
rINC2	increment sr (may repeat)
SUB_m_	subtract from memory byte/word through label (part 1)
SUB1n	subtract n from pr
SUBbpn	subtract n from memory byte through sr pointer
SUBwpn	subtract n from memory word through sr pointer

Table 21-3. Optimizer Generated P-codes

Take some time now to study Tables 21-1, 21-2, and 21-3 to become familiar with the p-code naming system. This will be time well spent. Learning the system will make reading the compiler's source files much easier.

A bit more explanation may help. The underscore symbol suffixes names that produce only the first part of an assembly language instruction or instruction sequence. The underscore should be read as though it were an ellipsis (...), meaning *etc.* It also prefixes names that complete an instruction (sequence).

The letter **r** indicates a repetition of whatever follows. The number of occurrences is indicated by the p-code's value. Thus,

```
rINC1
```

produces as many occurrences of the instruction that increments the primary register as the p-code's corresponding value indicates.

Some p-codes produce an assembler instruction that contains a numeric (compiler generated) label. The value associated with these codes designates the label number.

The P-Code Output Function

Associated with each p-code is a character string that specifies the ASCII data that the p-code should produce. These strings are related to the p-codes by means of an array of string addresses called **code[]**. Each p-code is a subscript into this array. The designated array element contains the address of the string

that translates the p-code. This arrangement makes for lightning-fast testing of p-codes when they are translated. **Outcode()** simply uses the p-code as a subscript into **codes[]**, then proceeds to process the indicated string.

Since Small C does not support the initializing of an array of pointers, the function **setcodes()** is called once, before parsing begins, to load **code[]** with its addresses. That function in **CC4.C** is the place to look for exactly what each p-code produces. It is very handy to have this association of p-code names and ASCII strings clustered at one place in the compiler. It saves a lot of searching around when we need to know the exact effect of a p-code.

As we saw in the description of **gen()**, several p-codes automatically trigger the generation of other codes. In these cases, we must also look at **gen()** to see the total effect of the original p-code. These p-codes are indicated with comments in **setcodes()**.

Translating p-codes to ASCII strings involves more than simply writing a string for a given p-code. The value associated with the p-code may influence the final form of the output in several ways. Thus, the p-code strings include a kind of "language" that directs **outcode()** in its application of the p-code's value. Table 21-4 lists the p-code translation strings.

Table 21-4. P-code Translation Strings

P-code	Translation
ADD12	\211ADD AX,BX\n
ADD1n	\010?ADD AX,<n>\n??
ADD21	\211ADD BX,AX\n
ADD2n	\010?ADD BX,<n>\n??
ADDbpn	\001ADD BYTE PTR [BX],<n>\n
ADDwpn	\001ADD WORD PTR [BX],<n>\n
ADDm_	\000ADD <m>
ADDSP	\100?ADD SP,<n>\n??
AND12	\211AND AX,BX\n
ANEG1	\010NEG AX\n
ARGCNTn	\000?MOV CL,<n>?XOR CL,CL?\n

P-code	Translation
ASL12	\011MOV CX,AX\nMOV AX,BX\nSAL AX,CL\n
ASR12	\011MOV CX,AX\nMOV AX,BX\nSAR AX,CL\n
CALL1	\010CALL AX\n
CALLm	\020CALL <m>\n
BYTE_	\000 DB
BYTEn	\000 DB <n>\n
BYTEr0	\000 DB <n> DUP(0)\n
COM1	\010NOT AX\n
COMMAn	\000,<n>\n
DBL1	\010SHL AX,1\n
DBL2	\001SHL BX,1\n
DECbp	\001DEC BYTE PTR [BX]\n
DECwp	\001DEC WORD PTR [BX]\n
DIV12	\011CWD\nIDIV BX\n
DIV12u	\011XOR DX,DX\nDIV BX\n
ENTER	\100PUSH BP\nMOV BP,SP\n
EQ10f	\010OR AX,AX\nJE $+5\nJMP _<n>\n
EQ12	\211CALL __EQ\n
GE10f	\010OR AX,AX\nJGE $+5\nJMP _<n>\n
GE12	\011CALL __GE\n
GE12u	\011CALL __UGE\n
GETb1m	\020MOV AL,<m>\nCBW\n
GETb1mu	\020MOV AL,<m>\nXOR AH,AH\n
GETb1p	\021MOV AL,?<n>??[BX]\nCBW\n
GETb1pu	\021MOV AL,?<n>??[BX]\nXOR AH,AH\n
GETb1s	\020MOV AL,<n>[BP]\nCBW\n
GETb1su	\020MOV AL,<n>[BP]\nXOR AH,AH\n
GETw1m	\020MOV AX,<m>\n
GETw1m_	\020MOV AX,<m>
GETw1n	\020?MOV AX,<n>?XOR AX,AX?\n
GETw1p	\021MOV AX,?<n>??[BX]\n
GETw1s	\020MOV AX,<n>[BP]\n
GETw2m	\002MOV BX,<m>\n

P-code	Translation
GETw2n	\002?MOV BX,<n>?XOR BX,BX?\n
GETw2p	\021MOV BX,?<n>??[BX]\n
GETw2s	\002MOV BX,<n>[BP]\n
GT10f	\010OR AX,AX\nJG $+5\nJMP _<n>\n
GT12	\010CALL __GT\n
GT12u	\011CALL __UGT\n
INCbp	\001INC BYTE PTR [BX]\n
INCwp	\001INC WORD PTR [BX]\n
WORD_	\000 DW
WORDn	\000 DW <n>\n
WORDr0	\000 DW <n> DUP(0)\n
JMPm	\000JMP _<n>\n
LABm	\000_<n>:\n
LE10f	\010OR AX,AX\nJLE $+5\nJMP _<n>\n
LE12	\011CALL __LE\n
LE12u	\011CALL __ULE\n
LNEG1	\010CALL __LNEG\n
LT10f	\010OR AX,AX\nJL $+5\nJMP _<n>\n
LT12	\011CALL __LT\n
LT12u	\011CALL __ULT\n
MOD12	\011CWD\nIDIV BX\nMOV AX,DX\n
MOD12u	\011XOR DX,DX\nDIV BX\nMOV AX,DX\n
MOVE21	\012MOV BX,AX\n
MUL12	\211IMUL BX\n
MUL12u	\211MUL BX\n
NE10f	\010OR AX,AX\nJNE $+5\nJMP _<n>\n
NE12	\211CALL __NE\n
NEARm	\000 DW _<n>\n
OR12	\211OR AX,BX\n
PLUSn	\000?+<n>??\n
POINT1l	\020MOV AX,OFFSET _<l>+<n>\n
POINT1m	\020MOV AX,OFFSET <m>\n
POINT1s	\020LEA AX,<n>[BP]\n

P-code	Translation
POINT2m	\002MOV BX,OFFSET <m>\n
POINT2m_	\002MOV BX,OFFSET <m>
POINT2s	\002LEA BX,<n>[BP]\n
POP2	\102POP BX\n
PUSH1	\110PUSH AX\n
PUSH2	\101PUSH BX\n
PUSHm	\100PUSH <m>\n
PUSHp	\100PUSH ?<n>??[BX]\n
PUSHs	\100PUSH ?<n>??[BP]\n
PUT_m_	\000MOV <m>
PUTbm1	\010MOV <m>,AL\n
PUTbp1	\011MOV [BX],AL\n
PUTwm1	\010MOV <m>,AX\n
PUTwp1	\011MOV [BX],AX\n
rDEC1	\010#DEC AX\n#
rDEC2	\010#DEC BX\n#
REFm	\000_<n>
RETURN	\100?MOV SP,BP\n??POP BP\nRET\n
rINC1	\010#INC AX\n#
rINC2	\010#INC BX\n#
SUB_m_	\000SUB <m>
SUB12	\011SUB AX,BX\n
SUB1n	\010?SUB AX,<n>\n??
SUBbpn	\001SUB BYTE PTR [BX],<n>\n
SUBwpn	\001SUB WORD PTR [BX],<n>\n
SWAP12	\011XCHG AX,BX\n
SWAP1s	\012POP BX\nXCHG AX,BX\nPUSH BX\n
SWITCH	\012CALL __SWITCH\n
XOR12	\211XOR AX,BX\n

Table 21-4. P-code Translation Strings

Probably, the most obvious thing about these strings is that each one begins with a byte that is coded with an octal escape sequence. That leading byte contains three codes that tell the optimizer about the effects of the instruction(s) in the string. It is not a part of the string proper, and so is skipped over when **outcode()** writes a string to the output file. More will be said about the meaning of these codes in Chapter 27.

Notice that these strings include occurrences of the **\n** escape sequence that specifies the newline character. **Outcode()** gives these no special treatment, so each one has its normal effect in the output file—it begins a new line. Since some strings have embedded occurrences of newlines, it follows that some strings produce multi-line instruction sequences—that is, multiple instructions. Notice too that some strings do not end with a newline character. These are the strings for the p-codes whose names end with an underscore, meaning that more text must follow to complete the instruction(s).

The "language" mentioned above, which tells **outcode()** how to apply the value associated with the p-code, consists of five devices (or *directives*). They are:

```
?...?...?
```

Question marks always appear in groups of three. Arbitrary text may occur between them. When **outcode()** sees the first question mark, it tests the value associated with the p-code for *true* or *false*. If true, the text between the first two question marks is written to the output file; otherwise, the text between the second two is written. As we can see from scanning Table 21-4, this directive is used in two ways. First, it selects between alternate forms of code so as to produce the most efficient instructions; for example, it is more efficient to zero a register by performing an exclusive OR of it with itself than by loading the constant zero. The second use is in deciding whether or not to generate optional instructions (or parts of instructions). This usually appears as **?...??**.

#...#

Number signs always occur in pairs. When **outcode()** sees the first number sign, it takes the p-code's value as a repetition count and writes the text between the number signs as many times as the count indicates. For instance, **#INC AX\n#** produces as many increment instructions as the p-code's value specifies.

<l>

A lowercase letter **l** (in angle brackets) tells **outcode()** to write the number of the current function's *literal pool label* as a decimal string. This is for references to string constants which reside in the literal pool that occurs at the end of each function. Each literal pool is preceded by a unique numeric label.

<m>

A lowercase letter **m** (in angle brackets) tells **outcode()** that the p-code's value is a pointer to a symbol table entry containing the name of a label which is to be used in a direct *memory* reference. On finding this, **outcode()** offsets the pointer by an amount that locates the symbol string in the table entry, and calls **outname()** to write the symbol to the output file as a name (in uppercase, preceded by an underscore).

<n>

A lowercase letter **n** (in angle brackets) tells **outcode()** that the p-code's value is a *number* that is to be written to the output file as a signed decimal string.

Outcode() does not write the actual special characters and code letters to the output file. They are replaced by the output that they designate.

At this point, the task of **outcode()** has been fully explained! It accepts a p-code and its value and it outputs the p-code's string while carrying out the operations indicated by these directives. It uses three integer locals to help it carry out its special directives.

Part indicates which part of a **?...?...?** directive is being processed. It contains a number indicating which question mark has been encountered.

Skip contains *true* or *false*, indicating whether or not the current part of the string is to be written to the output file. It is *true* during the part of a **?...?...?** sequence that is to be skipped.

Count counts down the number of repetitions of a **#...#** sequence. Also, two character pointers are used.

Cp is used to scan the string as it is being written.

Back saves the value of **cp** when it reaches the first text character in a **#...#** sequence. It is used later, when the terminal number sign is reached, to reset **cp** for the next repetition. When zero, it means that the next number sign will be the first one of a sequence rather than the last.

On entry, **outcode()** initializes **part** and **back** to zero, and **skip** to *false*. It then sets **cp** to the second character of the p-code's string; this is where the translation from p-code to string is accomplished. Next, **outcode()** falls into a loop which lasts as long as non-zero characters remain in the string. In the loop, it first looks for a **<**, indicating one of the lettered directives **<l>**, **<m>**, or **<n>**. On finding one, and if the current code is not being skipped, it tests the letter with a **switch** statement. Three cases cover the possibilities. Then **cp** is advanced over the directive and the loop continues.

That failing, **outcode()** looks for a question mark. If one is found and it is the first question mark in the sequence, it tests the p-code's value. If zero, it sets **skip** *true* so that the first text segment will be bypassed. On finding the second question mark, **skip** is logically negated. If skipping was going on, the second text segment is written to the output file, and vice versa. On finding the third question mark, **part** is reset, so the next question mark will be taken as the first one of a sequence, and **skip** is set to *false* so that the following text will be written normally.

If these tests fail, **outcode()** tests the current character for a **#**. Upon finding one, and seeing that it is the first one, it saves the p-code's value in **count** and **cp** in **back**. The loop then continues normally. Text is written until the second **#** is found. At that point, **count** is decremented and tested for more repetitions. If any remain, **cp** is reset to the address saved in **back** and the loop continues. This repeats until no more repetitions remain, at which time **back** is reset to zero and the loop continues.

If none of these special cases exists, **skip** indicates whether or not the current character is to be written. In either case, **cp** is advanced over the current character and the loop continues.

When the end of the string is reached, the loop terminates and **outcode()** returns, having translated the p-code and its value to assembly language in the output file.

CHAPTER 22

The Front End

This chapter shows how the source code is obtained, preprocessed, and scanned before going to the parser. We refer to these functions collectively as the *front end* of the compiler because they stand on the input side of the parser.

With most C compilers, the preprocessor is separate from the compiler itself. It makes its own pass on the input, changing its form for input to the compiler. With Small C, however, the preprocessor is integrated into the compiler so that it operates between the reading and scanning of the source code. This arrangement saves a pass on the program; but more importantly for Small C, it preserves the single pass design, which enables the compiler to respond immediately to program fragments.

Because of this integration and certain efficiency considerations, the distinction between preprocessing and parsing is blurred in the Small C compiler. For instance, two standard preprocessor directives, **#define** and **#include**, are handled by the parser, as are the Small C directives **#asm** and **#endasm**. Conversely, two of the scanning functions, **match**() and **streq**(), are also used by the preprocessor.

As we saw in Chapter 2, the term *token* refers to instances of the smallest language elements. Individual characters are not necessarily tokens, yet names, expression operators, constants, keywords, and punctuation marks are. It is important to realize that tokens may comprise one or more characters, and that they may or may not be separated by white space.

The Input Buffers

Before proceeding with the front end functions, it might help to review the data structures that they use. Recall from Chapter 20 that there are two input buffers—pointed to by **pline** and **mline**. Input is read, one line at a time, into **mline** from which it is preprocessed into **pline**.

The global pointer **line** points to one or the other of these in order to direct the scanning routines, which are shared by the parser and the preprocessor, to the correct buffer. The global pointer **lptr** points the scanning functions to the *current character* of **line,** and the global integer **ch** holds a copy of that character. This use of **ch** makes many references to the current character more efficient.

Three functions look for tokens in the input line: **symname()**, **match()**, and **amatch()**. These functions exhibit similar behavior in that they advance **lptr** past the current token if they find what they are looking for, but only if they find it. This allows the parser to make repeated attempts at recognizing the current token without passing it until it is finally accepted. These functions are described more fully below.

Front End Functions

Figure 22-1 diagrams the relationships among the major front end functions. Specifically, it shows the paths of control from the three scanning functions, at the top, down to the library function **fgets()** that reads lines from the current input file. A vertical line connecting two functions means that the higher function calls the lower function. Notice that **preprocess()** fits midway between the matching functions at the top and the I/O function at the bottom. Other, miscellaneous front end functions exist, but these suffice to show how program code filters from the input file up to the parser.

We look first at the miscellaneous functions, and then follow Figure 22-1 from top to bottom.

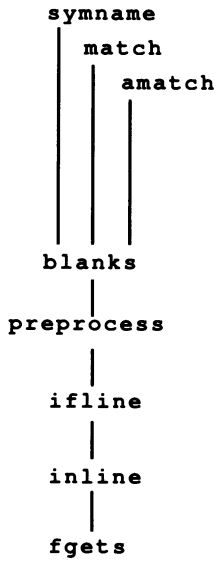


Figure 22-1: Major Front End Functions

Keepch() **Keepch**() serves **preprocess**() by placing one character at a time into **pline**. It first verifies that the character will fit. If not, it does nothing. In that case the character is lost, and when **preprocess**() reaches the end of the line it issues an error message.

Inbyte() **Inbyte**() returns the current character of the input line after advancing to the next one. It calls **gch**() to do this; however, it differs from **gch**() in that it calls **preprocess**() to fetch a new line if the end of the current one has been reached. Furthermore, if the new line is empty it goes for yet another, and so on until the next source character is found or the end of the last input file is reached. In the latter case, it returns zero.

Gch() **Gch**() returns the current character of the input line, advances **lptr** to the next one, and places it in **ch**. If the current character is the null terminator, then

gch() does not attempt to advance further, but does return the null character to the caller. **Gch**() calls **bump**() to advance to the next character.

Bump() **Bump**() either advances the current position in the input line (indicated by **lptr**) a specified number of positions beyond the current character, or it sets it to the beginning of the line. It accepts an integer **n,** which may be zero or a positive value. If **n** is zero, **lptr** is set to **line**; otherwise, it is increased by **n**. Then **ch** is assigned the value of the new character. Finally, if the new character is not the null terminator, **nch** (*next character*) is assigned the value of the following character; otherwise, it is assigned the null value. **Bump**() does not verify that **n** will not advance beyond the end of the line. It relies on the calling functions to ensure this.

Blanks() **Blanks**() advances the input past white space to the beginning of the next token or until the end of the input is reached. If necessary, it calls **preprocess**() to obtain new, preprocessed source lines. It calls **white**() to determine whether or not a character is to be skipped, and **gch**() to advance to the next character.

White() **White**() returns *true* if the current input character is a space or a control character and *false* otherwise.

Alpha() **Alpha**() returns *true* if the character passed to it is alphabetic or an underscore, and *false* otherwise.

An() **An**() returns *true* if the character passed to it is alphabetic, an underscore, or numeric, and *false* otherwise.

Streq() **Streq**() indicates whether or not the current substring in the source line matches a literal string. It accepts the address of the current character in the source line and the address of a literal string, and returns the substring length if a match occurs, and zero otherwise. While repeatedly matching characters from

left to right, if an inequality is found, failure is indicated. Otherwise, when the end of the literal is reached, success is indicated. Obviously, the end of the literal must determine the length of the comparison since the substring may be followed by anything and so cannot terminate the comparison.

Astreq() **Astreq**() indicates whether or not two alphanumeric strings or substrings match. It serves two purposes: to compare symbol names extracted from the source line with names in the symbol table, and to compare alphanumeric tokens in the source line with literal strings. In the latter case, it would seem to overlap the function of **streq**(), but there is a difference. **Streq**() will match a literal with the beginning of a token, whereas **astreq**() ensures that the entire token is examined. The term *string* is used now to refer to either an alphanumeric string with its own terminator, or an alphanumeric substring which is terminated by any non-alphanumeric character. **Astreq**() accepts three arguments—the addresses of two strings and the maximum number of characters to match on. It returns either the length of the matched strings or zero according to whether or not a match occurred. It loops, matching characters from the strings, from left to right until (1) inequality is found, (2) the end of the first string is reached, (3) the end of the second string is reached, or (4) the maximum length is exceeded. Then if the end of both strings are reached simultaneously, success is indicated; otherwise, failure.

Symname() **Symname**() is the first of the major front end functions in Figure 22-1. It is called whenever the parser thinks a name fits the syntax. It's purpose is to indicate whether or not a legal symbol is next in the input line and, if so, to copy it to a designated buffer and skip over it in the source line. **Symname**() first calls **blanks**() to advance past any white space at the current position in the line and, if necessary, to input and preprocess another line. If the first non-white character is not alphabetic (or an underscore), it returns *false* since that is a requirement for C names. Otherwise, it copies the symbol to the buffer pointed to by its only argument **sname**. If the symbol exceeds eight characters, only the first eight are copied and the rest are bypassed. Finally, it terminates the destina-

tion string with a null byte and returns *true*. **Symname**() calls **alpha**() to determine if the first character is alphabetic (or an underscore), **an**() to determine if other characters are alphanumeric (including underscore), and **gch**() to accept the current character from the input line and advance to the next one.

Match() **Match**() is one of two scanning functions that looks for a match between a literal string and the current token in the input line. It skips over the token and returns *true* if a match occurs; otherwise, it retains the current position in the input line and returns *false*. First, however, it calls **blanks**() to skip over white space to the next token, preprocessing a new line if necessary. It calls **bump**() to advance over the matched token. It is important to notice that since **match**() calls **streq**(), it matches the literal with the same number of characters in the source line and there is no verification that all of the token was matched.

Amatch() **Amatch**() is roughly equivalent to **match**() except that it assumes that an alphanumeric (including underscore) comparison is being made and guarantees that all of the token in the source line is scanned in the process. It uses **astreq**() to do the comparing.

Nextop() **Nextop**() is called by the expression analyzer to determine if the next token in the source line is one of a list of expression operators. The address of the string containing the list is received as an argument. Each operator in the string is separated from the others by a single space, and, as usual, the string is terminated with a null byte. With each iteration of an infinite loop, **nextop**() extracts into the local character array **op**[] the next operator from the list. It then calls **streq**() to match it to the current token. If there is a match, further tests are made to ensure that, for instance, < did not match <=. If a match did indeed occur, then *true* is returned. If the list is exhausted without a match, however, then *false* is returned. On success, the global integer **opindex** is set to the offset of the matched operator in the list—that is, its subscript. Later, in the expression analyzer, this will be adjusted so that it will correctly subscript the matched operator in the global arrays **op**[] and **op2**[].

Preprocess() **Preprocess()** is a rather large function that fetches and preprocesses a line of source code. It executes the following sequence of steps:

1. Set **line** to **mline**.
2. Call **ifline()** to obtain the next line that is not excluded by **#ifdef** or **#ifndef** directives.
3. Return immediately if the end of the last source file has been reached.
4. Copy **mline** to **pline** with special treatment given to *white space*, *character strings*, *character constants*, *comments*, and *macro names*.
5. Determine if **pline** has overflowed and, if so, issue the message "**line too long**."
6. Set **line** to **pline** for subsequent parsing.
7. Establish the first character as the current character for parsing.

The cases receiving special treatment are:

White space, of any length, is reduced to a single blank character.

Character strings are checked to ensure that there is a closing quote in the same line as the opening quote.

Character constants are checked to ensure that there is a closing apostrophe.

Comments are eliminated entirely from the preprocessed code. If necessary, additional lines are obtained until the end of a comment is reached.

Macro names are recognized and replaced by their substitution text. A token is suspected of being a macro name if it begins with an alphanumeric character (or underscore). It is then copied into a short character array and passed on to **search()** for a check against the macro name table. If it is not found in the table, it is simply copied directly to **pline**. If it is found, however, then its offset is taken from the table and used to locate the start of its replacement text in the macro text queue. From there, everything is copied to **pline** until a null character is reached. Finally, the remainder (beyond eight characters) of the macro name in **mline** is passed over.

Gch() is called to advance to the next character in **mline,** and **keepch**() is called to place characters into **pline**.

Ifline() **Ifline**() handles all matters pertaining to conditional compilation. Since it is called only by **preprocess**(), it should be viewed as part of the preprocessor. Standing between the source file(s) and **preprocess**(), it obtains source lines from **inline**(), looks for **#ifdef**, **#ifndef**, **#else**, and **#endif** directives, and decides the fate of the lines that they control—whether or not they are passed up to **preprocess**().

Ifline() contains one large infinite loop in which **inline**() is called at the top, followed by tests for the conditional compilation directives. If reached, a **break** at the bottom discontinues the loop and returns control to the preprocessor with a new line. If a conditional compilation directive is seen, it is duly noted and the loop is continued. Conditional compilation directives, therefore, never make it to the preprocessor.

The global integer **iflevel** serves to match each **#else** and **#endif** with its antecedent **#if...**. Each **#if...** increases it by one, and each **#endif** decreases it by one. Therefore, it reflects the nesting level of **#if...** directives.

Another global integer **skiplevel** indicates whether or not source lines are being skipped, and at what level the skipping was initiated. A segment of code being skipped is delimited by the next **#else** or **#endif** at the same level as the **#if...** that started the skipping.

The crucial statement in **ifline**() is

```
if(skiplevel) continue;
```

located before the **break** at the bottom of the loop. If **skiplevel** is not zero, the loop is continued and the current line is skipped. If it is zero, however, the **break** is reached and the current line is passed to the preprocessor.

At this point the following explanations should make sense:

An **#ifdef** advances **iflevel** by one, then checks **skiplevel** to see if it is currently skipping text. If so, it continues the loop. If it is not already skipping and if the specified symbol has not been defined, it should initiate skipping. Therefore, it calls **symname()** to extract the macro name from **#ifdef**, then it calls **search()** to look for the name in the macro name table. If the search *fails*, **skiplevel** is set to **iflevel** to initiate skipping and to record the nesting level at which it began. Finally, the loop is continued.

An **#ifndef** is handled exactly like an **#ifdef** except that skipping is initiated if the specified symbol *is* found in the macro name table.

An **#else** directive first checks **iflevel** to see if an antecedent **#if...** exists; that failing, the message "**no matching #if...**" is issued and the loop is continued. Assuming no such error, the **#else** must either initiate skipping, terminate skipping, or do nothing at all. It initiates skipping, regardless of its nesting level, if skipping has not already been initiated (**skiplevel** is zero). As with the previous two directives, this involves assigning **iflevel** to **skiplevel**. It terminates skipping only if skipping is already in progress and the nesting level matches the level at which skipping was initiated (**skiplevel** equals **iflevel**). This is done by assigning zero to **skiplevel**. Finally, if neither condition exists, the loop is continued without change.

An **#endif** directive also checks **iflevel** for an antecedent **#if...**, issuing "**no matching #if...**" if there is not one. Assuming no such error, the **#endif** must either terminate skipping, or do nothing at all. It terminates skipping only if skipping is already in progress and the nesting level matches the level at which skipping was initiated (**skiplevel** equals **iflevel**). As before, this is done by assigning zero to **skiplevel**. Finally, **iflevel** is decremented by one and the loop is continued.

One last check is made before returning to the preprocessor. If the new line is null, the loop is continued in an attempt to obtain a significant line.

Inline() **Inline()** fetches the next line of code from a source file and optionally lists it. Two global file descriptors direct it to input files; **input** designates

the current primary file, and **input2** designates the current **#include** file. (Since only one file descriptor is used for include files, nesting of include statements is not supported. See Chapter 28 for suggestions on making the compiler support nested include files.) Both descriptors are initially set to **EOF**, indicating that no input file has yet been opened. On entry, **inline()** checks **input** for **EOF** and, finding it, calls **openfile()** to prepare a file for input. Should that fail, **eof** would be set *true* and **inline()** would simply exit. Next, it decides which descriptor to use. A local integer **unit** is set to **input** or **input2** depending on whether or not **input2** equals **EOF**. **Unit** is then passed to **fgets()** to get the next line from the source file. In other words, if **input2** is not set to **EOF**, it contains the file descriptor of an active include file, and so it preempts the primary input file designated by **input**.

If the end-of-file condition is met, then the file designated by **unit** is closed, either **input** or **input2** (which ever pertains) is set to **EOF**, the input line is nulled by assigning zero to the first character, which is made current, and control is returned to the calling function. When this happens, a higher level function will eventually go for another line and **inline()** will again receive control. If the end-of-file occurred on **input**, **openfile()** will again be called to open the next source file, as described above. If it occurred on **input2**, however, then the include file will no longer preempt the primary input file and **input** will be used to fetch the next line from the current source file.

After successfully fetching a line, **inline()** tests **listfp** to see if a listing has been requested. If so, it writes the line to **listfp**. First, however, it checks to see if **listfp** is the same as the output file descriptor **output**. In that case, source lines are being interleaved with generated code, so each source line must be made to look like a comment by placing a semicolon before it.

Finally, before returning, **bump()** is called to establish the first character in the new line as the current character.

CHAPTER 23

The Main Function

Like all C programs, Small C begins execution in a function called **main()**. **Main()** resides in the first part of the compiler (file **CC1.C**) and performs six tasks: (1) displaying the signon message, (2) allocating storage for various buffers, (3) initializing variables, (4) setting run-time options, (5) opening files, and (6) controlling overall program flow.

Main(), and the other functions mentioned below, can be located in the compiler listings (Appendix C) by referring to the function index in Appendix H. Textual references are also given so that further investigation of these functions can be easily pursued.

Displaying the Signon Message

The file **NOTICE.H** contains macro definitions for two symbols which specify version and copyright information. **VERSION** specifies the version and revision numbers of the compiler, and **CRIGHT1** contains copyright information. **NOTICE.H** is included into part one of the compiler, making these notices available to **main()** for display on the console.

Allocating Storage

Calloc() is called repeatedly to allocate the switch queue, the staging buffer, the while queue, the literal pool, the macro name and text buffers, the line buffers, and the symbol table. This process also involves saving the addresses of these data structures in pointers and placing buffer-ending addresses in other pointers.

Initializing Variables

Most global objects are initialized either by default or by the use of initializers. **Main()** initializes others that must be set at run time. Those are **argcs**, **argvs**, **locptr**, and **glbptr**. Of course, the setting of pointers to allocated blocks of memory, described above, is also an initializing activity.

Processing Command-Line Switches

Command-line switches are processed by **ask()**. Basically, this consists of setting default values for several global variables, then scanning the command line for switches that specify different values for them. These variables are described below:

Alarm determines whether or not to sound an audible alarm when an error is reported. It is initially zero by virtue of its static storage class. If **ask()** finds **-A** in the command line, it sets **alarm** *true*.

Monitor determines whether or not to display the header lines of function declarations. **Monitor** is initially zero by virtue of its static storage class. If **ask()** finds **-M** in the command line, it sets **monitor** *true*.

Optimize determines whether or not to perform the peephole optimizing operations. It is initially set *true*. If **ask()** finds **-NO** in the command line, it sets **optimize** *false*.

Pause determines whether or not to pause after an error is reported. It is initially zero by virtue of its static storage class. If **ask()** finds **-P** in the command line, it sets **pause** *true*.

If **ask()** finds an undefined switch, it displays a usage message showing which switches are valid, and then aborts the run. This reminder can be evoked deliberately by supplying the null switch (**-**).

To gain access to the command-line arguments, **ask()** calls the Small C library function **getarg()**. Any arguments without a leading hyphen are considered to be filenames and so are ignored by **ask()**. By converting switch characters to uppercase and comparing them to uppercase constants, **ask()** is effectively case blind.

Opening Files

Before compiling begins, **main**() calls **openfile**() to open the first source file that might be specified in the command line (also an output file of the same name but with an **ASM** extension). Later, **openfile**() will be called, at a lower level, to look for the next source file when the current file has been exhausted.

Like **ask**(), **openfile**() calls **getarg**() repeatedly, looking for file names (any argument not preceded by a hyphen). That failing, it sets **input** to the value of **stdin**, thereby establishing the default input device. (Recall that the standard input file may be redirected to any file or input device.) On the other hand, if it finds a filename, **openfile**() tries to open it for input. First, however, it supplies C as a default extension if none was given. If an input file was given, **openfile**() assumes that the output should go to a file of the same name, but with an extension of **ASM**—unless the standard output file was redirected from the screen, that is. Redirecting it implies that the user has other ideas, and so the output should be left alone. If **openfile**() cannot open a file, it displays an error message and aborts the run.

After opening the first input file, **main**() calls **preprocess**() to fetch the first input line for parsing.

Controlling Program Flow

With these preliminaries out of the way, **main**() calls five functions to effect high-level control over the compiler. In order of execution, they are as follows:

Header(), as we saw in Chapter 21, writes preliminary text to the output file.

Setcodes() initializes the array **code**[] to the addresses of the p-code translation strings. It also calls **setseq**(), which initializes **seq**[] to the addresses of arrays of optimizing sequences. **Peep**() scans this array to find potential optimizing cases (Chapter 27).

Parse() is the highest level parsing function. Chapter 24 describes **parse**(); suffice it to say here that this step in **main**() is where the compiling takes place.

Trailer() writes text at the end of the output file. This consists of (1) external declarations for functions that were referenced but not declared, (2) the external

declaration of **_main()** (if the module being compiled contains the function **main()**), (3) a directive that terminates the active memory segment, and (4) the **END** directive that signals the end of the output file. Also, if the compiler was compiled with **DISOPT** defined in CC4.C, this function dumps optimizing statistics to the standard output file.

Finally, **fclose()** is called to close the output file. This call is not strictly necessary since all open files will be closed upon program exit anyway.

After that, **main()** exits through its closing brace and control returns to the operating system.

CHAPTER 24

High-Level Parsing

A parser might employ any of a number parsing methods of which there are two broad categories—*top-down* methods and *bottom-up* methods. For a comparative study of parsing methods, you are referred to references in the Bibliography under the heading "Compiler Writing." Specifically, I suggest Chapter 6 of Calingaert [19] for a quick survey, and Tremblay and Sorenson [24] for more detail.

Small C uses a popular top-down parsing method called *recursive descent.* If a program is successively divided into smaller and smaller components, it forms a tree based on the grammar of the language. At the root is the entire program; at the interior nodes are instances of constructs like statements, expressions, and functions; and at the leaves are tokens.

The recursive descent method starts at the root of such an imaginary tree for the program and looks for a sequence of global constructs. When it recognizes the beginning of one, it then tries to recognize the constituent parts. These, in turn, are broken down into smaller parts, until the smallest lexical units (the tokens) are reached.

In the case of Small C, this activity is accomplished by having a high-level function **parse**() look for the global constructs. **Parse**(), in turn, calls (*descends to*) lower-level functions that recognize the next most inclusive constructs, and so on. When the bottom (token level) is reached, control returns back up to the lowest level at which a construct was recognized; from there it descends again to find the next part of the construct. When the construct has been fully parsed, control migrates up to the next higher level at which a construct was recognized, and then down again. As the parser "walks" this imaginary tree, it emits *object code* that represents the program in a lower level "target" language. This process

continues until control reverts back to the highest level and no more global constructs remain.

To repeat, the parse tree for the program is only imaginary—it does not exist as such; the parser only performs a linear scan of the source program. On the other hand, instantaneous points on this imaginary tree do have a real existence in the compiler in the form of nested stack frames for the parsing functions that are active at any point in the process.

Recursion enters the picture because it is implicit in the grammar of the language. For example, a statement might be a compound statement, containing other statements. Therefore, the parsing function **statement()** might call the lower-level function **compound()**, which in turn calls **statement()**.

The various function calls and returns effectively travel down and up the branches of the parse tree for the program being compiled. Generally speaking, movement down the tree (toward the leaves) corresponds to successful efforts to recognize language constructs, and movement up the tree corresponds to the generation of code based on what has been recognized.

Top-Level Parsing

Parse() Small C begins parsing in **parse(),** which recognizes the global declarations (data and functions) that constitute a complete program. In addition, as mentioned in Chapter 22, it also recognizes the **#include**, **#define**, and **#asm** directives. **Parse()** hangs in a loop until **eof** is *true*—the end of the last source file has been reached. With each iteration it attempts to recognize the current token as the start of a declaration (**int**, **char**, **unsigned**, **extern**) or one of the directives (**#asm**, **#include**, or **#define**). Should these fail, it assumes it has a function definition. Each of these constructs is further parsed by one of the second-level functions indicated in Figure 24-1. These five functions parse the entire program.

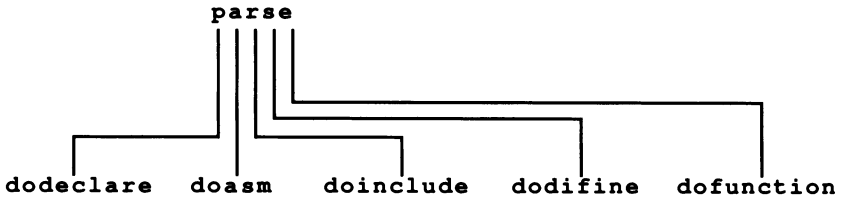


Figure 24-1. Primary High-level Parsing Functions

Unlike the other four functions, **dodeclare()** is called in anticipation of a declaration. If it finds one, it parses it and returns *true*; not finding one, it returns *false*. It handles both external and static declarations. If **parse()** sees the keyword **extern,** it calls **dodeclare()** passing it the constant **EXTERNAL** as a signal that the external storage class applies; otherwise, it passes **STATIC** for the static storage class.

Recall from Chapter 22 that **amatch()** and **match()**, which recognize literal strings in the source, take care of numerous low-level, front-end tasks. These include bypassing white space, bypassing recognized tokens, reading and preprocessing new source lines, and even closing one source file and opening the next one if necessary. They return *true* only when the specified string is recognized. **Amatch()** differs from **match()** by assuming that an alphanumeric comparison is being made on an entire token—the token must exactly match the string and the first character after the token must not be alphanumeric, whereas **match()** is satisfied if the string matches only the beginning of the token, regardless of what follows.

Parsing Global Data Declarations

Dodeclare() **Dodeclare()** is a small function that tries to match on **char**, **int**, **unsigned**, **unsigned char**, or **unsigned int**. Finding none of these, it assumes **int** if the storage class is **EXTERN**; this agrees with normal C syntax in which the keyword **int** may be omitted following **extern**. Failing these conditions, control

returns to **parse()**, indicating *false*. If a declaration is recognized, however, **dodeclare()** calls **declglb(),** passing it the data type (**CHR**, **INT**, **UCHR**, or **UINT**) and the storage class. After that, it calls **ns()** to enforce the terminal semicolon, and returns *true*.

Declglb() **Declglb()** parses global declarations (see Listing 19-2 for examples). On entry from **dodeclare()**, the optional storage class (**extern**) and the data type (**char**, **int**, **unsigned**, **unsigned char**, or **unsigned int**) have already been recognized, skipped over, and passed to this function that now finds itself facing a list of names to declare and possibly define (reserve storage and initialize).

Each iteration of an infinite loop processes the next name in the list. Two statements test for return conditions. At the top of the loop

```
if(endst()) return;
```

calls **endst()** to test for the end of the statement (semicolon or end of input) and returns if *true*. And, at the bottom,

```
if(match(",") == 0) return;
```

returns when a declaration is not followed by a comma. Between these, **declglb()**:

1. Determines the identity (**POINTER**, **VARIABLE**, **ARRAY**, **FUNCTION**) of the object, saving it in the local variable **id**.

Note: Small C recognizes either the * prefix (for **POINTER**) or the [suffix (for **ARRAY**), but not both. It cannot handle an array of pointers. If both are specified, an error message is issued.

2. Determines the number of occurrences of the object, saving it in the local variable **dim**. This may be greater than one for an array.
3. Fetches and verifies the object's name.
4. Verifies that the name has not already been declared—does not yet exist in the symbol table.
5. Uses **id** and **dim** to generate appropriate code to define the object if it is

not a function. Predeclared functions are only added to the symbol table.

6. Uses **id** and **dim** to add an entry to the symbol table so the object can be referenced.

First, a check is made for *, indicating **POINTER**. That failing, the assumption is **VARIABLE**. Then, **symname()** is called to verify the name and copy it from the source line into **ssname[]**. Next, **findglb()** is called to verify that the name is not yet in the symbol table. After that, if the identity is **VARIABLE**, checks are made for [or (, indicating **ARRAY** or **FUNCTION**, respectively. These override the initial determination of **VARIABLE**. Notice that these checks are not made if the name was previously identified as a pointer. In that case, if these tokens are present, the loop terminates after declaring the pointer, and an error results when **dodeclare()** tries to enforce the terminal semicolon.

The function **needsub()** is called to evaluate the dimension expression following [, and to enforce the presence of a closing]. Its returned value is the specified dimension—a constant.

Next, if the storage class is **EXTERNAL**, **external()** (Chapter 21) is called to make the declaration to the assembler. Otherwise, if the identity is not **FUNCTION** (i.e., data is being declared), **initials()** (see below) is called to define and initialize the object(s). Finally, **addsym()** (also below) is called to enter the declared object into the global symbol table.

Initials() **Initials()** is passed the size of the object (**type>>2**), its identity (**POINTER**, **VARIABLE**, or **ARRAY**), and the number of occurrences (**dim**) to define. Internally, these are called **size**, **ident**, and **dim,** respectively. First, the declared symbol name, in **ssname[]**, is declared to the assembler as an entry point by calling **public()**. This guarantees that it can be reached from every module of the program. Next, **initials()** scans for

```
= Value
```

or

```
= {Value, Value, ..., Value}
```

where each ***Value*** is evaluated by **init()**. The values are stored in the literal pool until the end of the initializer is reached, and then dumped to the output as either **DB** or **DW** directives. After that, to account for the fact that the number of initial values may not be as great as the dimension of an array, **dumpzero()** is called to fill out the remaining elements with zeroes. This means that an array will be the size of the larger of its dimension or the number of initial values it has. **Initials()** can be called with **dim** equal to zero under two conditions—a pointer is being declared or an array of dimension zero is being declared. Therefore, after calling **init()** (which decrements **dim** for each initial value), a check is made to see if **dim** started at zero and is still zero. If so, an uninitialized pointer or an uninitialized and undimensioned array is declared. In the first case, **initials()** assumes a value of zero; in the second, it complains with "**need array size.**"

Init() **Init()** recognizes two types of initializer—character strings and constant expressions. There are restrictions on which type is legal for a given object (see Table 7-1), so **init()** detects violations and reports them.

First, **init()** calls **string()** to test for a quoted string. Finding one, **string()** (1) saves it in the literal pool (giving **init()** the starting offset as a side effect in the local integer **value**), (2) skips the closing quote, (3) terminates the string in the pool with a null byte, and (4) returns *true*. Otherwise, it returns *false*.

In fetching the characters from the string, **string()** calls **litchar()** to translate escape sequences to individual characters. If a string is found and the object being initialized is not a character array or character pointer, the message "**must assign to char pointer or char array**" is issued. Next, the dimension variable in **initials()** is reduced by the length of the string (the new literal pool offset **litptr** minus the original offset **value**). Finally, if the object being defined is a pointer, then space for it is allocated and initialized to the address of the string. In this case, since the string will be dumped immediately following the pointer,

```
DW $+2
```

is generated by calling **point()**.

If a quoted string is not found in the input line, **constexpr**() is called to look for a constant expression. On finding one, a test is made to see if the object being defined is a pointer; if so, the message "**cannot assign to pointer**" is issued. In any case, the constant value is stored in the literal pool and the dimension (in **initials**()) is reduced by one.

Parsing Preprocessor Directives

Doasm() **Doasm**() is a small and very simple routine for handling assembly language code. It differs from the other second-level parsing functions in that it may be called from **parse**() or **statement**(). This is because Small C allows assembly code to appear not only at the global level, but anywhere a statement is allowed as well. When called, the lead-in **#asm** has already been recognized and bypassed, so **doasm**() merely drops into a loop copying lines of input to the output until a line beginning with **#endasm** is reached. Of course, the end-of-input condition also terminates the loop. Finally, the terminating line is discarded by calling **kill**(), and control returns to the caller.

Doinclude() **Doinclude**() is also a very simple function. When called, the keyword **#include** has already been recognized and bypassed, and a filename, delimited by quotation marks or angle brackets, is next up (note that Small C also accepts it without delimiters). **Doinclude**() skips the leading quote or angle bracket, if present, and copies the filename into the local character array **str[]** until the trailing delimiter or the next white space is reached. Then it calls the library function **fopen**(), passing it **str** for the filename and **"r"** (retrieval) for the mode. If the open is successful, **input2** is assigned the new file descriptor, thereby causing **inline**() to fetch future input from that file instead of the primary file indicated by **input**. Should the open fail, however, **input2** is again assigned its normal value of **EOF**, and "**open failure on include file**" is issued. Finally, **kill**() is called to discard the current line so that the next token will have to come from the new file.

Dodefine() **Dodefine**() is another small routine. It processes **#define** directives by adding the macro name to the macro name table and then copying the remaining text into the macro text queue. When called, the keyword **#define** has been recognized and bypassed, and the macro name is next up. **Symname**() is called to verify that the name is legal and to fetch it into **msname**[]. If it is not a valid name, "**illegal name**" is issued, the line is discarded, and control returns. Otherwise, **search**() is called to look for the name in the macro name table. If not found, **cptr** is left pointing to an available entry into which the name is copied. In case the name is not found and there is no more space available, **cptr** is set to zero and "**macro name table full**" is issued before control returns. Regardless of whether a new entry was established or an existing one was found, the entry designated by **cptr** is pointed to the location in **macq** where the new replacement text will be stored—in the first case a new macro is being defined and in the second case an old one is being redefined. Finally, the remaining text in the line, starting with the next token, is copied into the text queue where it is terminated by a null byte. If there is insufficient room, "**macro string queue full**" is issued and the run aborts.

Parsing Function Definitions

As mentioned before, if **parse**() does not recognize any of the lead-in keywords for declarations or the directives which it handles, it assumes that a function is being defined, and so it calls **dofunction**(). This large function:

1. Initializes the literal pool, obtains a label number for the literals that will be dumped at the end of the function, and clears the local symbol table.
2. Bypasses the optional keyword **void** if there is one.
3. Lists the current line on the screen if monitoring was requested.
4. Verifies that the current token (the function name) is a legal symbol and fetches it into **ssname**[]. That failing, "**illegal function or declaration**" is issued, the line is discarded, and control returns to **parse**().
5. Determines whether or not the symbol is already in the global symbol table. If so, the current definition might be an illegal redefinition. This would be

the case if the table entry is not a function or is a function and was created by a previous definition. Conversely, if it was created by an earlier function reference (recognized by a storage class of **AUTOEXT**), the current definition is proper and the class is changed to **STATIC**.

Note: Functions that are referenced, but not defined, are automatically declared external at the end of the program by **trailer()**; hence the name **AUTOEXT**.

If no entry matching the current function name is found in the table, one is created with:

```
NAME   =    ssname
IDENT  =    FUNCTION
CLASS  =    STATIC
TYPE   =    INT
SIZE   =    0
OFFSET =    0
```

6. Generates the string in **ssname[]** as a label and entry point by calling **public()**.

7. Requires the opening parentheses for the argument list. Failure to find it yields "**no open paren.**"

8. Declares each formal argument in the local symbol table. This is done in a loop looking for the closing parenthesis (also terminated abnormally by the **endst()** condition). With each iteration:

a. The current token is determined to be a legal name and is fetched into **ssname[]**; otherwise, "**illegal argument name**" is issued and the illegal name is bypassed.

b. If the name is not already in the local symbol table, it is added with:

```
NAME    =    ssname
IDENT   =    0
CLASS   =    AUTOMATIC
TYPE    =    0
SIZE    =    0
OFFSET  =    0, 2, 4, ...  for the 1st, 2nd, 3rd, argument
```

If it is in the table already, **multidef()** issues "**already defined.**" Note that **IDENT** and **TYPE** cannot be established until the declarations following the closing parenthesis are parsed.

c. If the following token is not a right parenthesis or a comma, "**no comma**" is issued.

9. Determines the identity and type of each declared argument. This is done in another loop in which the variable **argstk**—used above for assigning **OFFSET** values—counts down to zero. With each iteration, a declaration list is processed by calling **doargs(),** which decrements **argstk** by two for each argument it parses. Before each call, **int**, **char**, **unsigned**, **unsigned int**, or **unsigned char** must be recognized to introduce the declaration list (and a semicolon afterward). If one of these keywords is not found, "**wrong number of arguments**" is issued since, because **argstk** is not zero, not every argument has been typed and something other than a declaration has been found.

Each call to **doargs()** accepts the data type from **dofunction()** and determines the identity and size of a list of arguments. The function **decl()** is called to determine these.

Decl() is also used for local declarations. In this case, however, since the argument **aid** (array id) has the value **POINTER**, arrays are assigned identities of **POINTER**. This makes sense because their addresses are passed to the function on the stack, making them lvalues. Whether **decl()** is handling argument or local declarations, it requires that functions have the syntax of a pointer; otherwise, "**try (*...)()**" is issued. **Decl()** issues "**illegal symbol**" if an improper name is found.

Doargs() issues "**not an argument**" if a name in the list cannot be found in the local symbol table. Having found the argument's entry in the local symbol table, **doargs()** sets the **IDENT**, **TYPE**, **SIZE**, and **OFFSET** fields to complete the declaration. As we saw, it gets **IDENT** and **SIZE** from **decl()**, and it receives **TYPE** as an argument. It inverts the **OFFSET** field (to account for the fact that Small C pushes actual arguments onto the stack from left to right) and adjusts it to account for the presence of a return address and the saved value of BP on top of the stack when the function receives control. For example, if three arguments are being passed, step 8b. would have established their **OFFSET**s as 0, 2, and 4. Then this step would invert and adjust them to 8, 6, and 4 respectively. From this point on, arguments are treated just like local variables. The only difference being that they are placed on the stack before the call (by the calling function), whereas local variables are allocated on the stack after the call (by the called function).

10. Calls **statement()**. This is the central event in **dofunction()** because **statement()** parses the entire body of the function! More will be said about this important step in Chapter 25.

11. Generates the default return that passes control back to the caller when it reaches the end of the function body. This is done only if the last statement in the function is not a **return** or **goto**; otherwise, the default return could never be reached.

12. Finally, if any literal strings were defined within the function, they are dumped from the literal pool to the output. This involves generating the label that was reserved in step (1) (and referenced in the function body) and generating the necessary **DB** directives.

Now that we have surveyed the steps that declare a function, in the next chapter we will look deeper into the heart of the compiler by examining **statement()** and its subordinate functions.

CHAPTER 25

Low-Level Parsing

At this point in our tour of the Small C compiler we investigate the third- and fourth-level parsing functions—the ones that handle statements and local declarations. This is where the big picture begins taking shape. Figure 25-1 illustrates how the major low-level parsing functions relate to each another.

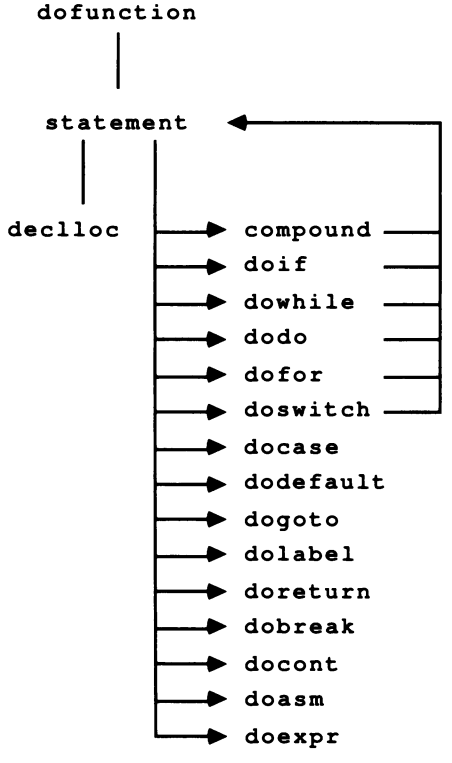


Figure 25-1. Primary Low-Level Parsing Functions

As we saw in Chapter 24, the high point in **dofunction()** is where it calls **statement()**, since that is where the entire body of a function is parsed. This should seem appropriate since the body of a function is really just a single compound statement.

Statement() in turn calls the lower-level functions shown in Figure 25-1. **Declloc()** declares local variables. **Compound()** handles compound statements, while **doif()**, **dowhile()**, **dodo()**, **dofor()**, and **doswitch()** parse **if**, **while**, **do**, **for**, and **switch** statements, respectively. These six functions are special because they each call **statement()** recursively. Notice that each of these constructs controls one or two dependent statements. And these, in turn, might contain further instances of these six control statements. This nesting could go on to any depth. It will be important to remember that this recursion exists not only when studying these six functions, but when studying all of the functions (except **dofunction()**) in Figure 25-1 since they all can be reached through recursion.

As with the preceding six functions, the others parse the constructs that their names imply. **Doasm()** was described in Chapter 24 and so is not repeated here. Neither is **doexpr()**, since it comprehends the expression analyzer that is reserved for Chapter 26.

Third-Level Parsing

Statement() Each call to **statement()** parses a single statement. Sequences of statements only occur within a compound statement (or block), so the loop that recognizes statement sequences is found in **compound()**.

First, **statement()** looks for a data declaration, introduced by **char**, **int**, **unsigned**, **unsigned char**, or **unsigned int**. Finding one of these, it calls **declloc()**, passing it the data type (**CHR**, **INT**, **UCHR**, or **UINT**). It then enforces the presence of a semicolon.

If that fails, it looks for one of the tokens that introduces an executable statement. If that succeeds, the corresponding parsing function is called, and the global variable **laststst** is set to a value indicating which statement was parsed. Table 25-1 lists these keywords, their parsers, and their **laststst** values.

TOKEN	PARSER	LASTST
{	compound	
if	doif	STIF
while	dowhile	STWHILE
do	dodo	STDO
for	dofor	STFOR
switch	doswitch	STSWITCH
case	docase	STCASE
default	dodefault	STDEF
goto	dogoto	STGOTO
:	dolabel	STLABEL
return	doreturn	STRETURN
break	dobreak	STBREAK
continue	docontinue	STCONT
;		
#asm	doasm	STASM
		STEXPR

Table 25-1. Statement Tokens, Parsers, and Lastst Values

Several cases deserve special comment. First, no value is assigned to **lastst** following a compound statement. This is because **lastst** must represent the last elementary statement that is guaranteed to have executed when control reaches the end of the function. Assigning a value after a compound statement would wipe out the desired information. Recall from Chapter 24 that **lastst** is tested by **dofunction()**, after parsing a function's body, to decide whether or not to generate a default return. At first glance, it would appear that assigning a value to **lastst** after each statement would not give reliable results. For instance, a function might end with

```
...
if(a==b) return;
}
```

If this should leave **laststst** with the value **STRETURN**, then the default **RET** would not be generated and the consequences would be disastrous. However, that does not happen because the **if** statement recursively calls **statement()** to parse the **return**. Therefore, as the recursion unwinds, **laststst** is first assigned **STRETURN** and then **STIF**.

Labels are handled differently since the colon which identifies them is not a prefix token, but a suffix. In this case, **dolabel()** is called to determine whether or not a label is present and, if so, to handle it. It returns *true* on success.

Null statements—semicolons—are also special because no parsing is needed. **Errflag** is reset, however, to reenable error reporting if it was disabled previously. Small C has a policy of reporting only the first error message for a simple statement since the cascade of messages that usually follow are more likely to be spurious than correct. In practice, this works quite well.

If no other statement is recognized, it is assumed that the current token is the beginning of an expression, and so **statement()** calls **doexpr()**. Recall that in C, an isolated expression is a legal statement. If it is not clear at this point how a function is called, then remember that functions are called from within expressions.

Fourth-Level Parsing

Declloc() **Declloc()** parses local declarations (see Listing 19-6 for examples). This function is roughly parallel to **declglb()**, although major differences exist. First, local declarations may appear only at the beginning of a block, before the executable statements. This means that all of the local variables in a block can be allocated on the stack at one time with:

```
ADD SP,-n
```

in which ***n*** is the total number of bytes being allocated. If a data declaration is encountered after an executable statement, "**must declare first in block**" is issued.

Since local variables are referenced relative to the base pointer BP, no labels are generated. Rather, the local symbol table contains the offsets from BP to the variables.

Another difference is found in the fact that local variables are not initialized, and so initializers do not have to be parsed. Also, Small C restricts local function declarations to pointers; that is, only function declarations of the form:

```
(*name)()
```

are acceptable at the local level. These factors further simplify the task of **declloc()**.

Two other restrictions on local declarations need to be considered. First, local declarations are not allowed in the body of a **switch** statement. This is because control enters a **switch** statement at arbitrary points without passing through the head of a block where the local variables would be allocated. Thus, it would be possible to reference variables that were never allocated. Therefore, attempting to declare locals within a **switch** statement yields the message "**not allowed in switch**." An example of such an illegal statement is:

```
switch(i) {
  int j;
  case 1: j = i; ...
  ...
  }
```

Control cannot reach the point, before the first case prefix, where **j** is allocated space on the stack.

The other restriction has a similar basis. **Goto** statements and locals (declared below the highest-level block) are not allowed in combination—either is allowed, but not both. This is because jumping into a block in which locals are declared would bypass the code that allocates the locals. Local declarations at the highest level in the body of a function present no problem since they precede all executable statements, and so are guaranteed to be allocated. A violation of

this restriction is greeted with either "**not allowed with goto**" or "**not allowed with block-locals.**" An illegal example is:

```
func() {
  if(i) {
        int j;
        target: j = 0;
        ...
        }
  goto target;
  }
```

On entry from **statement()**, the data type (**char**, **int**, **unsigned**, **unsigned char**, or **unsigned int**) has already been recognized, bypassed, and passed to this function as an argument. **Declloc()** then finds itself facing a list of objects to define (reserve storage for). Each iteration of an infinite loop processes the next object in the list. Two statements test for return conditions. At the top of the loop,

```
if(endst()) return;
```

tests for the end of the statement (semicolon or end of input) and returns if so. And at the bottom,

```
if(match(",")==0) return;
```

returns when an object declaration is not followed by a comma. Between these, **declloc():**

1. Calls **decl()** to determine the identity (**POINTER**, **VARIABLE**, **ARRAY**) and size of the object, saving them in the local variables **id** and **sz**. **Decl()** also fetches and verifies the object's name.
2. Uses **id** and **sz** to add an entry to the symbol table so the object can be referenced.

First, within **decl**(), a check is made for an open parenthesis, introducing a function pointer declaration—(*...)(). Next, * is sought; if present, the identity is set to **POINTER**; otherwise, **VARIABLE** is assumed. **Symname**() is then called to verify and fetch the name. If a legal name is not there, **illname**() complains. Next, if an open parenthesis was found, the closing parenthesis is bypassed. Then, if another open parenthesis is found, a function pointer is being declared. In that case, failure to supply the leading (* produces "**try (*...)()**" as an error message. In any case, the final closing parenthesis is required.

Next, if the identity is **VARIABLE** (not **POINTER**), then a check is made for an open bracket, indicating **ARRAY**. This overrides the initial assumption of **VARIABLE**. **Needsub**() evaluates the dimension expression and enforces the closing bracket. Its returned value is the specified dimension—a constant. If the dimension is missing or evaluates to zero, then "**need array size**" is issued. The array dimension is multiplied by the size of an element to determine the total size of the array. Last of all, the size of the declared object is returned to **declloc**().

Finally, **declloc**() calls **addsym**() to enter the declared object into the local symbol table.

Notice that nothing is done to actually generate code to allocate the declared object. That step is postponed until the first executable statement is reached. The global variable **declared** accumulates a running total of the number of bytes that have been declared—the number that must be allocated. Notice that it gets reset to zero by **compound**() at the beginning of a block. Then as each object is declared, it gets incremented by the size of the object. Finally, when **statement**() sees something other than a declaration, the statement

```
gen(ADDSP, csp-declared);
```

generates code to adjust the stack so as to reserve enough space for the variables. It also sets **declared** to -1 as a signal to **declloc**() that further declarations are to be rejected with the message "**must declare first in block.**"

Compound() **Compound**() parses compound statements. The essence of this function is captured in the lines:

```
++ncmp;
while(match("}") == 0)
  if(eof) {
        error("no final }");
        break;
        }
  else statement();
if(-ncmp
&& laststst != STRETURN
&& laststst != STGOTO)
gen(ADDSP, savcsp);
```

The global variable **ncmp** indicates the level of nesting of compound statements. Notice that it gets increased before the statements between the braces are parsed, and decreased afterward. On entry, the opening brace has already been recognized and bypassed, so the next thing should be the first statement. The loop continues until the closing brace or the end of input text is reached. Since the program should not end in an unclosed block, "**no final }**" is issued in that case. With each iteration of the loop, **statement**() is called recursively. Notice that, since **statement**() calls **compound**(), any of these inner statements may itself be compound, causing other instances of this same loop to be activated within a single iteration of this one. A closing brace will terminate the innermost instance of the loop, thereby properly matching the last open brace—a happy consequence of recursion.

What happens before and after the main loop? First, **csp** and **locptr** are saved so they can be restored after the block is parsed. The statements:

```
savcsp = csp;
savloc = locptr;
```

accomplish this. Also, **declared** is set to zero so that local declarations will be enabled.

After parsing the block, the stack may require adjusting to deallocate any locals that were declared, the local symbol table is purged of symbols that were declared in the block, and **declared** is set to -1 to ensure that locals cannot be declared until another block is opened.

There is no need to generate stack restoring code, however, if the last unconditional statement in the block is a **return** or **goto**. In the first case, the stack adjusting code is generated before the **return** and in the second it is not needed because it cannot be reached. So **laststst** is checked and,

```
gen(ADDSP, savcsp);
```

is performed only if necessary. Whether or not code is generated to deallocate locals, **csp** is restored to its original value for further parsing.

It would be tempting to purge locals from the symbol table simply by means of:

```
locptr = savloc;
```

That would purge everything declared in the current block and leave everything declared in superior blocks. It would wipe out labels along with objects, however, and that would not do since labels must be seen throughout a function's body. Therefore, before restoring **locptr**, a sweep from **savloc** to **locptr** finds each label entry, moves it to **savloc** and advances **savloc** over the relocated entry. Only then is:

```
locptr = savloc;
```

allowed to discard local objects that were declared in the current block.

Doif() This function and the next three are very similar. They do essentially the same things but in different sequences. We shall examine this one very closely and then simply describe differences in the others.

The task of **doif()** is to scan an **if** statement from left to right, generating assembly code that faithfully represents the statement to the CPU. Figure 25-2 il-

lustrates the general form of the C and assembly code for both forms of the **if** statement. We shall make one pass through **doif()**, observing how it handles both types of statement—with and without an **else** clause.

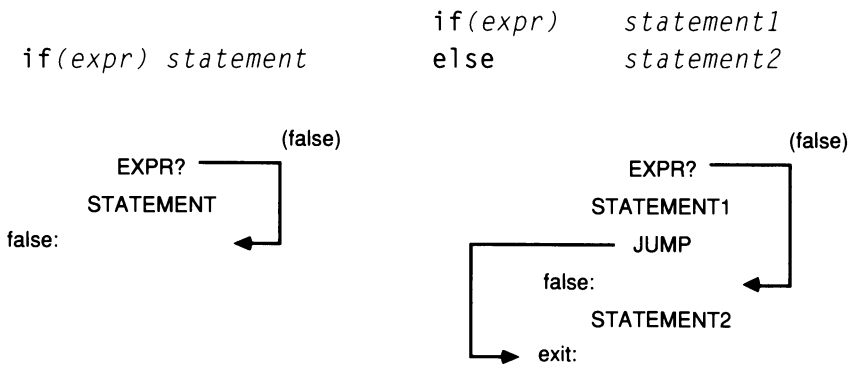


Figure 25-2. Structure of IF and IF/ELSE Statements

First, a label number must be reserved for use in jumping around the first (or only) controlled statement. In Figure 25-2, this label is designated **false:**, although its actual appearance in the output file is ***_n:***, where ***n*** is the number of the label. The statement

```
test(flab1 = getlabel(), YES);
```

obtains a unique label number for the false label, assigns it to **flab1**, and calls **test()** to scan the expression (or list of expressions) in parentheses and generate code to evaluate it and test the result. This code, which appears as **EXPR?** in Figure 25-2, contains a jump on the *false* condition to the false label. If the expression evaluates *true*, however, the jump is not taken and control falls down to whatever follows. Since the next thing in the C syntax is the statement that should execute under the *true* condition, that statement is parsed by calling **statement()** to generate code for it (**STATEMENT** and **STATEMENT1** in Figure

25-2). Since it closely relates to expression analysis, the examination of **test()** has been postponed to Chapter 26.

Now, if the **else** clause is not present, we only have to generate the **false** label with:

```
gen(LABm, flab1);
```

and return.

If an **else** clause is present, however, there must be another call to **statement()**. But we don't want the second statement to execute after the first one, so a jump from the end of the first statement around the second one is needed. Therefore,

```
flab2 = getlabel();
```

reserves another label number for this purpose and:

```
gen(JMPm, flab2);
```

generates the jump. Notice, however, that the jump is not always generated. It is not needed if the *true* statement (or the last statement in it) is a **return** or a **goto**. In either case the jump could never be reached, so a bit of optimizing is done here. Now the false label is generated by:

```
gen(LABm, flab1);
```

and the second statement is parsed by calling **statement()**. Finally, the exit label is generated by:

```
gen(LABm, flab2);
```

and the **else** clause has been parsed.

Notice that calls to **statement**() are recursive, so any legal statement—even another **if**—could occur at that point. Try mapping out, as in Figure 25-2, the form of the code generated by

```
if(expr1)
  if(expr2) statement1
  else statement2
else statement3
```

to show that the **else** clauses are correctly matched with their antecedents.

As we saw, this is a very simple, straightforward function. It is not difficult to understand if we know what the underlying functions do. This function is typical of the next three; essentially, they differ only in order and complexity, but not in concept.

Dowhile() Figure 25-3 illustrates the general form of the C and assembly code for the **while** statement.

```
while(expr)  statement
```

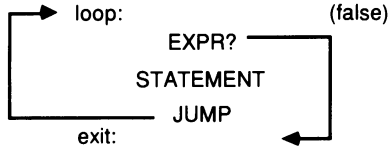


Figure 25-3. Structure of a WHILE Statement

Although this function is smaller and more straightforward than the previous one, it does introduce calls to **addwhile**() and **delwhile**(). These were discussed in Chapter 20 with the while queue.

Addwhile() establishes a new entry in the while queue to represent the present level of nesting of this control structure. It contains the stack level on entry to the current statement, the number of a target label for continuing the loop, and the number of a target label for breaking out of the loop. These three elements are also stored in **wq[]**—a local copy of the new while queue entry. Next, the loop label (**loop**: in Figure 25-3) is generated by:

```
gen(LABm, wq[WQLOOP]);
```

Then,

```
test(wq[WQEXIT], YES);
```

generates code to evaluate the expression and test the result (**EXPR?** in Figure 25-3). This contains a jump on the *false* condition to the exit label. After that, **statement()** is called recursively to parse the controlled statement. This is followed by an unconditional jump to the loop label. Finally, the exit label is generated and the current entry in the while queue is deleted.

As we can see in Figure 25-3, the generated code first evaluates the expression in parentheses. If it tests *true*, the controlled statement is executed and the process repeats. When the expression tests *false*, control goes to the exit point and continues on with whatever follows.

Dodo() Figure 25-4 illustrates the general form of the C and assembly code for the **do** statement.

```
do statement  while(expr);
```

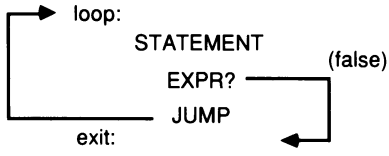


Figure 25-4. Structure of a DO Statement

There is very little difference between this function and **dowhile()**. Essentially, the order of the expression evaluation and the controlled statement are reversed. The statement is executed first and then the condition is tested. Besides that, there is just the additional requirement of the token **while**.

Dofor() Figure 25-5 illustrates the general form of the C and assembly code for the **for** statement.

```
for(expr1; expr2; expr3) statement
```

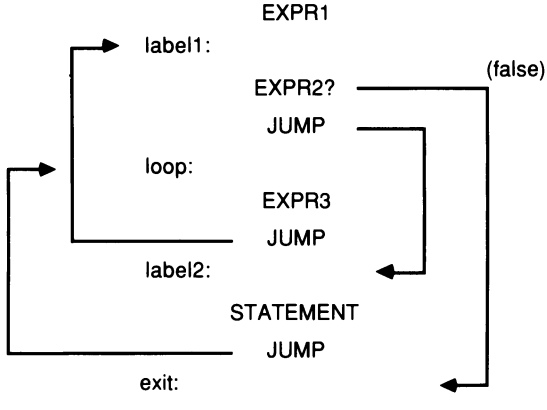


Figure 25-5. Structure of a FOR Statement

This function is larger and more complex than the previous ones, but it is really no more difficult to understand since it contains nothing that we have not already seen. By way of reminder, the proper order for the performance of a **for** statement is:

1. Evaluate the first expression.
2. Evaluate and test the second expression for the exit condition.
3. Execute the controlled statement.
4. Evaluate the third expression.
5. Go back to step 2.

As we can see from Figure 25-5, the generated code does just that. Two of the unconditional jumps are necessary because the parts of the **for** statement are not executed in the order of their appearance in the syntax. Specifically, the third expression is evaluated after the controlled statement. Since it precedes the statement in the syntax, however, Small C generates the code for it before the code for the statement. The first two jumps are needed to keep the chronology of events straight.

Any of the three expressions may be omitted as long as the two semicolons are present. Notice what happens when they are absent. If the first one is missing, there is no code generated for it and the statement begins with the evaluation of the second expression. If the second one is missing, there is no code generated for it or for testing it; and, since the *false* jump is not there, the missing expression is always interpreted as *true*. Finally, if the last expression is missing, no code is generated for it so, after the controlled statement executes, the last jump transfers control to the second one which immediately transfers it to the first one. It should be clear why an infinite **while** loop is preferable.

Doswitch() Figure 25-6 illustrates the general form of the C and assembly code for a **switch** statement.

This is easily the most devious statement that has to be parsed. Since **__switch** plays a crucial role in the execution of **switch** statements, reviewing its description in Chapter 18 may be in order at this point.

As with the other statements, code for the various parts of the **switch** statement is generated in the order of appearance with jumps inserted to keep the order of execution straight. Thus, the expression under test is evaluated first, leaving its value in the primary register (AX). Next, a jump skips around the controlled statements to call to **__switch**. This routine compares AX to the table of label/value pairs following the call. Each label corresponds to a case statement; its corresponding value is the value of the case expression. The table is terminated with a word containing zero, which cannot possibly be an address. When a match is found—AX equals a case value—a jump is executed from **__switch** directly to the corresponding label. This is shown in the figure as a jump from a table entry to the target label. If the end of the table is reached without finding a match, **__switch** simply returns control to the address following the end of the table. In our example, since there is a **default** statement, a jump to the default label was generated immediately after the table, so control goes there. Had there been no **default** statement, the jump would not have been generated, and control would have gone to the exit label from which it would have continued to whatever follows.

```
switch(expr) {
        case 15: statement1;
        case 12: statement2; break;
        default: statement3;
        }
```

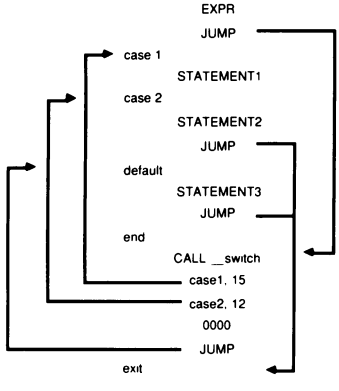


Figure 25-6. Structure of a Switch Statement

Notice that since there is no **break** following **statement1**, both of the first two statements execute if the expression evaluates to 15. The first of the two jumps to the exit label is generated by the **break** statement, but the last one is always generated to make control exit properly at the end of the last statement.

With one exception, this function uses the while queue like the previous ones. Since **switch** statements do not have a continuation point, specifying zero for a loop label prevents a **continue** statement from seeing this entry in the queue.

Everything between braces (top of Figure 25-6) is generated by a single recursive call to **statement()** since the body of a **switch** is really just an ordinary compound statement in which **case**, **default**, and **break** statements are allowed.

Since nesting is possible, there must be multiple instances of three variables—**swactive**, **swdefault**, and **swnext**. When *true*, **swactive** allows **case** and **default** statements to be accepted. When not zero, **swdefault** contains the label number of the **default** prefix. And **swnext** points to the next available switch table entry. These three variables are declared globally, but are saved locally on each entry to **doswitch()** and restored on exit. Therefore, they always contain values that are appropriate to any given level of nesting.

Recall from Chapter 20 that the purpose of the switch table is to store label/value pairs that are created by **case** statements. They must be retained until the end of the body of the **switch** is reached, at which point they can be generated in the output. Because of nesting, pairs from several different nesting levels may be present in the table at one time. However, the lowest level entries are at the end of the table from which they are released as unnesting occurs. By restoring **swnext** to its value when **doswitch()** was entered, only those entries at the lowest level are released.

Docase() **Docase()** is a small function. Basically it does three things: it generates a label for the **case** statement, it evaluates the constant expression associated with the **case** statement, and it places both in the next available entry of the switch table. As we saw, these label/value pairs accumulate in the table until **doswitch()** dumps them at the end of the **switch** statement.

Four errors are caught. If **swactive** is zero, "**not in switch**" is issued. If the switch table overflows, "**too many cases**" is issued. If the **case** expression does not yield a constant, "**must be constant expression**" is issued. And if the terminating colon is not found, "**missing token**" is issued.

Dodefault() Basically, this function only generates a default label and saves its number in the global variable **swdefault**, telling **doswitch()** to generate (after the label/value table) a jump to the default label.

Three errors are detected. If **swactive** is zero, "**not in switch**" is issued. If **swdefault** is already non-zero, "**multiple defaults**" is issued. And if the terminating colon is not found, "**missing token**" is issued.

Dobreak() **Dobreak()** generates a jump to the exit label of the innermost **while**, **for**, **do**, or **switch** statement as indicated by the last entry in the while queue. First, it calls **readwhile()** to obtain a pointer to the last entry in the while queue. If the queue is empty, "**out of context**" is issued and the jump is not generated since there is no active control statement. That failing, code to adjust the stack back to its original value is generated, after which a jump to the exit label is also generated.

Docont() **Docont()** does the same thing as **dobreak()** except for two differences. First, it targets the loop label instead of the exit label. Then, since the **switch** statement does not have a loop label, this function must search backward through the while queue for the last entry with a loop label number. This is done by

```
while(1) {
  if((ptr = readwhile(ptr)) == 0) return;
  if(ptr[WQLOOP]) break;
  }
```

Since **readwhile()** returns the address of the entry preceding the one passed to it, each iteration backs up one entry until an entry with a loop label number is found or **readwhile()** issues "**out if context**" and returns zero.

Dogoto() **Dogoto**() is a simple function that generates a jump to a designated label in the current function. If the target label name is legal, **addlabel**() is called to add the name to the local symbol table and associate it with a label number. Since the target label may precede or follow the **goto** statement, the label may or may not be in the table already. If it is, **addlabel**() does nothing; otherwise, it adds the label to the table. In either case, it returns the label number which **dogoto**() passes to **gen**() as the target label number. With this arrangement, the programmer's label name never appears in the output. A corresponding numbered label acts as a synonym. This prevents "**multiply defined**" errors from being issued by the assembler when more than one function contains the same label name.

Notice that there is no attempt to adjust the stack before executing the jump. This is because Small C, by disallowing local declarations except prior to executable statements (and labels), guarantees that the stack level will be uniform throughout the function. This exclusion is initiated by the occurrence of the first **goto** in the function. On the other hand, if a declaration in a nested block appears first, then **goto** statements are disallowed. In other words, the two are mutually exclusive; whichever appears first is allowed and the other is excluded.

Four errors are possible. If local variables have been declared in a nested block, "**not allowed with block-locals**" is issued. If the label name is found in the local symbol table, but the entry is not for a label, "**not a label**" is issued. If the token following the keyword **goto** is not a legal symbol "**bad label**" is issued. And if a semicolon does not terminate the statement, "**no semicolon**" is issued.

Dolabel() **Dolabel**() generates labels at the points where they appear in the function. This function must first decide whether or not the current token in the input line is a label. It begins by passing over white space to locate the token in question. It then saves **lptr**, the source line pointer, so that if the token is not a label, **lptr** can be restored to prevent further scanning from missing the current token. If the token is not a legal symbol, it is not passed over and *false* is returned to **statement**(). If it is a legal symbol, however, it is copied into **ssname**[] and

bypassed in the source line. If the next token is a colon, then the extracted symbol is taken for a label. **Addlabel()** is called to add it to the symbol table (if it is not already there), the numeric form of the label is generated in the output, and *true* is returned. On the other hand, if the symbol is not followed by a colon, **bump()** is called to move the current position back to the start of the token, and *false* is returned.

Doreturn() Doreturn() **recognizes a** return **statement and generates**

```
MOV SP,BP
POP BP
RET
```

in the output file. The first instruction is generated only if the **return** occurs within the scope of local variables. In that case, SP must be adjusted back to its value before the first local object was allocated. Also, **csp** must be adjusted back to its global value of zero. All of this is done by

```
gen(RETURN, 0);
```

Before that, however, the return expression, if present, is evaluated by calling **doexpr()** (Chapter 26). If no expression is there, no code is generated. In either case, whatever is in AX becomes the return value of the function.

Notice that **csp** is preserved across the generation of the return code; this is because the return may be executed conditionally, and so more code may follow. In that case, the stack level on entry to the **return** statement applies to the following code. If the return is not taken, SP is not adjusted and **csp** is preserved.

CHAPTER 26

Expression Analysis

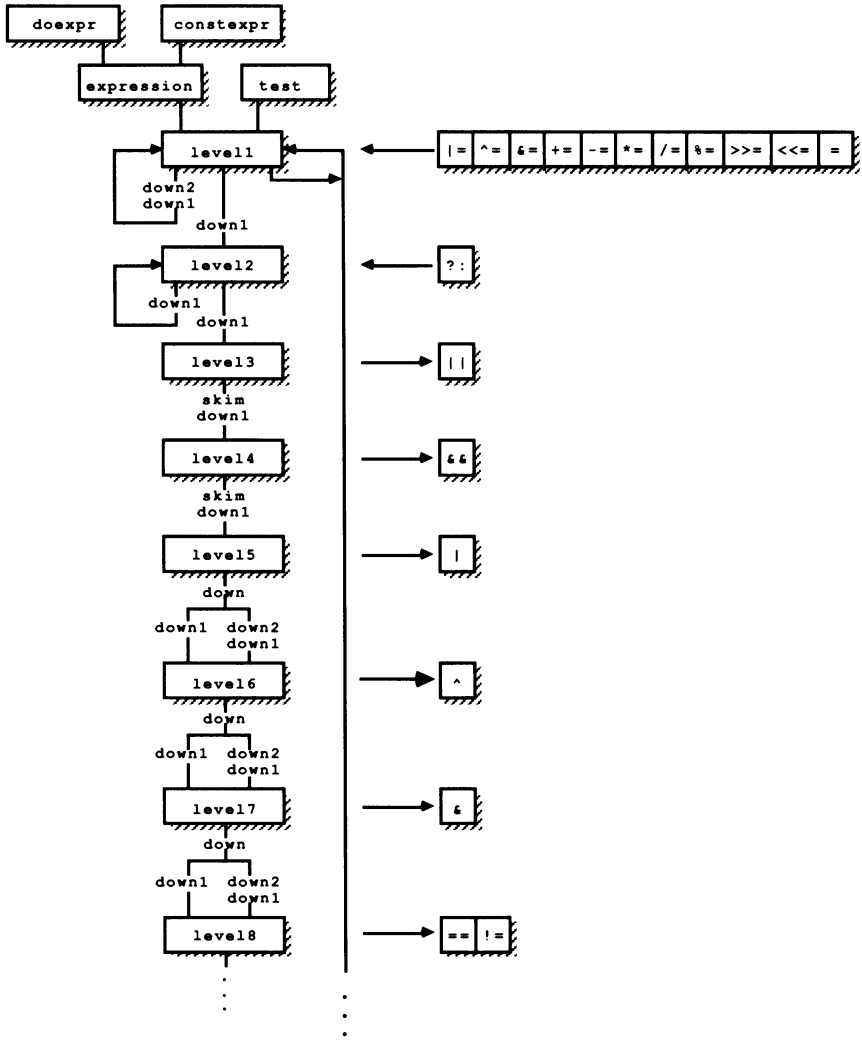
We come now to the last remaining task: the parsing of expressions. This is clearly the most difficult part of the compiler. Therefore, it has been separated from the previous chapter for special treatment.

To help us appreciate the overall organization and operation of the expression analyzer, Figure 26-1 presents a graphic view of the major analyzer functions, the operators they recognize, and the grouping of the operators. On the left side are the functions; the most important ones (in terms of the flow of control through the analyzer) are in boxes for emphasis. The basic flow of control is from the top down. To save space, in many cases the connecting lines have been squeezed out. So, where we see function names stacked above one another, there is an implied call from the upper to the lower function. Recursion is evident where lines of control go from a function back to itself or to **level1()**.

Functions **level1()** through **level14()** correspond to the precedence levels of the expression operators, such that **level1()** recognizes the operators at the *lowest* precedence level and **level14()** the *highest*.

To the right of each function, enclosed in boxes, are the operators that the function recognizes. Each set of operators is preceded by an arrow showing the way the operators associate. Notice that the functions for the only three levels that associate from right to left (**level1()**, **level2()**, and **level13()**) perform recursive calls to themselves. This causes the analyzer to scan its way to the rightmost instance of a run of operators at that level (lower level operators being parsed along the way) before generating code for these operators as the recursion unwinds from right to left.

At the bottom of Figure 26-1 is the function **primary()** that recognizes the operands upon which the operators work. These appear as identifiers (variables,



(continued on next page)

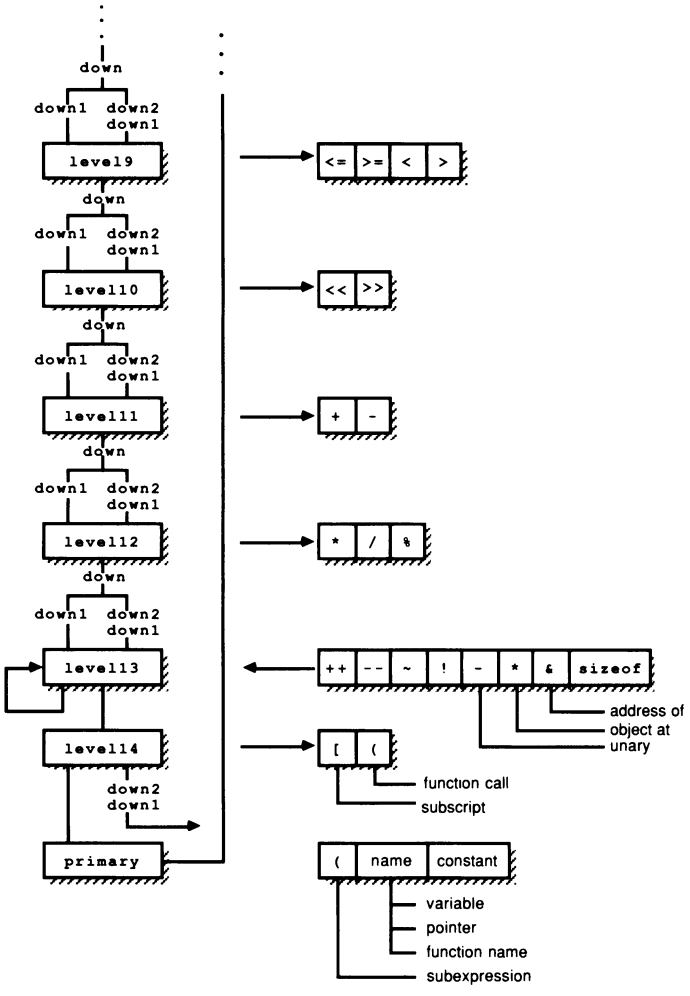


Figure 26-1. Expression Analysis Functions

pointers, and functions), constants, or subexpressions in parentheses. A function name stands for the address of the function. If it is followed by a left parenthesis, it is recognized by **level14()** as a function call; otherwise, its address figures into the expression. Small C is unique in that it interprets an undeclared symbol to be a function name and places it in the global symbol table as such.

As we saw in Chapter 25, there were several points in the syntax where expressions had to be parsed. For example, the **if** statement has an expression (or list of expressions) in parentheses that must be evaluated and tested at run time. Therefore, **doif()** contained

```
test(flab1 = getlabel(), YES);
```

to parse the expression and generate code for its evaluation and testing. As we can see from Figure 26-1, **test()** is one of three lead-in functions for the expression analyzer. Another is **constexpr()**, which is called when a constant expression, such as a **case** value or an array dimension, is expected. There is also **doexpr(),** which attempts to interpret an entire statement as an expression.

Now, what about those unboxed functions—**skim()**, **down()**, **down1()**, and **down2()**? Basically, they contain logic which used to be imbedded in the **level** functions. While that arrangement used less stack space and was somewhat faster, it made the compiler considerably larger by replicating virtually identical logic many times. So this version of Small C has extracted this redundant logic into these *pipeline* functions through which most levels of the analyzer call other levels.

An Introductory Example

Before launching right into a study of the analyzer, it should help to first get a feel for the overall flow of control by tracing out the parsing of a sample expression. Consider the program:

```
int i, j, k;
main() {
  i = j+k/5;
  }
```

In this case, the expression is a complete statement, so we enter the analyzer from **statement()** through **doexpr()** and **expression()**, at the top of Figure 26-1. Before anything else, **level1()** calls **level2()** through **down1()** which remembers the staging buffer location in case a constant results. In that case, on return, it would discard everything that was generated by lower functions, on the assumption that a higher function will replace it with a single statement that puts the constant in the primary register.

Level2() immediately calls **Level3()** by way of **down1()**. **Level3()** then immediately calls **skim()** which in turn immediately calls **level4()** through **down1()**. **Level4()** then, likewise, calls **level5()** through **skim()** and **down1()**.

Level5() is the first of eight functions which are virtually identical. It immediately calls **level6()** by way of **down()** and **down1()**. This process continues all the way down to **level13()**.

Notice that until now there has been no attempt to scan the expression. In this function, however, since unary operators may precede an operand, attempts are made to match on any of the seven unary operators (Figure 26-1). That failing, **level13()** calls **level14()** directly. **Level14()** then immediately calls **primary()** which (1) sees the symbol **i** in the input line, (2) skips over it, (3) finds it in the global symbol table, (4) passes its table address and data type (**int**) up the line as a side effect, and (5) returns *true* (meaning that **i** has yet to be fetched).

At this point, the calls begin to unnest as the **level** functions, in reverse order, attempt to recognize the operators to which they are sensitive. As each fails, it returns to the next higher level the value that originated in **primary()**. As each preceding instance of **down1()** regains control, it merely returns the value from **primary()** up the line. Likewise, each instance of **down()** (which is empowered by the **level** function above to recognize and act on selected operators) does nothing but return the value from **primary()** up the line. A quick look at the listing for **level5()** through **level12()** makes it obvious that all they do is return the value (in this case *true*) that they receive. This unwinding continues up through levels four and three with the pipeline function **skim()** finding nothing to do.

Finally, control arrives back in **level1()** where the assignment operator (=) is recognized and bypassed. **Level1()** then takes note of the table address and the

data type of **i** and proceeds to call itself; this is because other assignment operators could appear to the right of this one.

Now we descend all the way down the ladder again and in **primary()** recognize **j**. Again, control runs back up the ladder. Only this time, since the next token is a plus sign, something different happens in the instance of **down()** called by **level11()**. It recognizes the plus sign. And, seeing that **j** has yet to be fetched, it generates code to load it into AX. It then bypasses the plus sign and calls **level12()** again, but this time through **down2()** and **down1()**. **Down2()** first generates code to push AX (containing **j**) onto the stack. This preserves it until whatever lies to the right of the plus sign has been evaluated. Then it can be popped into BX and added to the right side.

Again, control steps down to **primary()** where, this time, **k** is recognized. On the way back up, the following token (a slash), is recognized by the instance of **down()** beneath **level12()**. Seeing that **k** has yet to be fetched, it generates code to load it into AX. It then bypasses the slash and calls **level13()** again, but this time through **down2()** and **down1()**. Before calling **down1()**, however, **down2()** generates code to push AX (containing **k**) onto the stack. This preserves it until whatever lies to the right of the slash has been evaluated, at which time it can be popped and divided by the right side.

Once again, control goes down to **primary()** where the number **5** is recognized as a constant. It is converted to an internal integer and passed as a side effect back up the line. Finally, **primary()** returns *false* meaning that this operand has already been fetched.

At this point, the constant **5** is being passed to higher functions, the values of **j** and **k** are on the stack (more correctly, they would be at run time), and the next token is the semicolon that terminates the statement. As control runs back up the ladder, it reaches the instance of **down2()** beneath **level12()** where the value **5** (received as a side effect from **primary()**) is seen as a constant. And, since BX is not needed to evaluate the constant, the code that pushes **k** onto the stack is purged from the staging buffer, and in its place:

```
MOV BX,AX
MOV AX,5
```

(p-codes **MOVE21** and **GETw1n**) is generated. Now, with **5** in AX and **k** in BX, **down2**() generates:

```
XCHG AX,BX
CWD
IDIV BX
```

(p-code **DIV12**) which performs the division operation, leaving the quotient in AX and the remainder in DX. (The **XCHG AX,BX**, which is automatically generated by **gen**() when it sees **DIV12**, swaps the operands so they will be in the appropriate registers for the 8086 divide instruction. The **CWD** instruction converts the dividend in AX to a double word by extending its sign through DX. The **IDIV BX** instruction performs a signed divide of DX,AX by BX.)

After this, control works its way up to the instance of **down2**() beneath **level11**() where code is generated that pops **j** from the stack into BX. Now with **k/5** in AX and **j** in BX, **down2**() generates:

```
ADD AX,BX
```

(p-code **ADD12**) which adds BX to AX, placing the sum in AX.

Next, control propagates all the way up to **level1**() again, where an attempt to recognize further assignment operators fails and control returns to the previous instance of **level1**() that had called itself. Here code is generated to store AX in memory where **i** resides, thereby effecting the assignment operation.

Finally, **level1**() returns to **expression**() which returns to **doexpr**() and the analysis is finished. The result is:

```
MOV AX,_J       fetch j into AX
PUSH AX         push j on the stack
MOV AX,_K       fetch k into AX
MOV BX,AX       move k to BX
MOV AX,5        load 5 into AX
XCHG AX,BX      swap AX and BX
CWD             sign extend AX into BX
IDIV BX         divide k (DX,AX) by 5 (BX)
```

```
POP BX          pop j into BX
ADD AX,BX       add j (BX) to k/5 (AX)
MOV _I,AX       store j+k/5 (AX) into i
```

which carries out the desired expression evaluation.

It is interesting to see what happens when the default order of precedence is changed by parentheses as in

```
i = (j+k)/5
```

In this case, after scanning the equal sign, control runs down the line to **primary()** where the left parenthesis is seen, causing it to directly call **level1()** for the evaluation of **j+k**. Then, since the right parenthesis is not recognized, control reverts back up to **level1()** which, thinking it is finished, returns. This time **primary()** receives control, however, enforces the right parenthesis and returns control back up the ladder, causing analysis to continue. The result is:

```
MOV AX,_J       fetch j into AX
PUSH AX         push j onto the stack
MOV AX,_K       fetch k into AX
POP BX          pop j into BX
ADD AX,BX       add j (BX) to k (AX)
MOV BX,AX       move j+k (AX) to BX
MOV AX,5        load 5 into AX
XCHG AX,BX      swap 5 (AX) and j+k (BX)
CWD             sign extend AX into DX
IDIV BX         divide j+k (AX) by 5 (BX)
MOV _I,AX       store (j+k)/5 (AX) in i
```

Notice how the placement of higher-precedence operators toward the bottom of the evaluation hierarchy and the lower-precedence operators toward the top ensures that operations are performed in the proper order. Also notice how easy it is to allow parentheses to alter the default order by making **primary()** sensitive to them. We simply treat anything between matching parentheses as an entire expression within the context of a larger expression.

Let's draw from this overview some additional concepts. First, notice that the unary operators (level 13) that precede operands are recognized on the way down the hierarchy, while those that follow are recognized on the way up. By "on the way down" I mean before calling the next lower level, and by "on the way up" I mean after returning from the lower level. This is obvious after thinking about it a bit, yet it helps to keep this in mind as we study the analyzer functions. Analogously, the binary operators are recognized on the way up from scanning the left operand; however, since they require an operand on the right side, they must descend from the present level back down the hierarchy to evaluate the right-hand operand. Then, on return from that, code is generated to effect the binary operation before returning to the next higher level.

This behavior is reflected in Figure 26-1 by the repeated pattern of the **down** functions connecting levels 5 through 13. **Down()** starts the descent to the next lower level. **Down1()** suffices as the pipeline for the left operand, and **down2()** followed by **down1()** serves to handle the right operand if an operator at the current level is recognized. So, the main path is the left path, with the right path being used only to descend one level after a binary operator has been detected. As a mnemonic device, these paths are arranged in Figure 26-1 on the left and right under **down()** according to which operand is being sought. As we shall see later, the mechanism by which information is passed—through side-effects—from **primary()** to higher functions must be duplicated so that information about both operands entering into a binary operation is available to **down2()** for analysis.

The Lead-in Functions

Having established a feel for the flow of control, we now look into the actual functions which constitute the analyzer. First, we examine the lead-in functions **doexpr()**, **constexpr()**, and **test()**.

Doexpr() **Doexpr()** is called when **statement()** cannot recognize a keyword—the assumption being that it must have an expression statement. **Doexpr()** is a small function that does just two things. In case there is a string of expressions, it checks for a comma after every call to **expression()** and, finding

one, loops back for another call. It also manipulates the staging buffer by calling **setstage()**, to set the staging buffer to its initial position, before **expression()** is called. Afterward it calls **clearstage()** to optimize and dump the buffer to the output file.

Constexpr() **Constexpr()** is called when the syntax demands a constant expression. This is often simply a constant—numeric or character—but it may be any legal expression resulting in a constant value. Unlike **doexpr()**, this function does not expect or handle a list of expressions, so it only looks for one. Like **doexpr()**, however, it sets and clears the staging buffer. There is a major difference, though. When it calls **clearstage()**, it passes a zero for the second argument, meaning that the staging buffer is to be cleared but not dumped. In other words, the code generated by analyzing the expression is discarded.

This seemingly strange behavior is understandable when we consider two things. First, constant expressions must sometimes be evaluated for compile-time use (e.g., array dimensions), in which case it would be inappropriate to generate code for run time execution. And, even when the expression occurs in a run-time context, it is more efficient to generate a single instruction to load the constant value than to generate a set of instructions that perform a stepwise evaluation of the expression.

So, **expression()** has been written to return two items as side effects: the value of the expression and either *true* or *false* depending on whether or not that value is a constant. **Constexpr()** tests the first of these to see if the expression was indeed constant as expected. If it was not, then "**must be constant expression**" is issued. Finally, it returns the value of the expression. Of course, this is a bogus value if the message was issued.

Expression() **Expression()** is one of two functions that initiate expression analysis by calling **level1()**. Besides calling **level1()** and returning the two items mentioned above, it does one very important thing. It declares locally an array of seven integers into which **level1()** and subordinate functions place vital information about the expression. This is the first instance of the soon-to-be-familiar **is[]** and **is2[]** arrays. These arrays are the means by which, starting with **pri-**

mary(), information is returned by side effects up the hierarchy. The term *is* in these array names is not an acronym, but the word. These arrays tell the analyzer what the entire expression or any part of it *is*. In other words, they carry the properties of whatever has been parsed at every point in the process.

This first instance of the array is for the left leg of the descent through all of the levels. Another (**is2[]**) is declared—for use with the right operand—at each level where a binary operator is recognized. Before control rises to a higher level, the *left* array is adjusted to reflect the result of the operation, and the *right*, being local and having served its purpose, is discarded. At lower levels, the *right* array appears as a *left* array, and other *right* arrays are declared as other binary operators are encountered. Figure 26-2 illustrates the points where these arrays are declared during the analysis of the expression:

```
i=j+k/5;
```

It shows a parse tree for the expression with the *left* and *right* arrays written in where they are created.

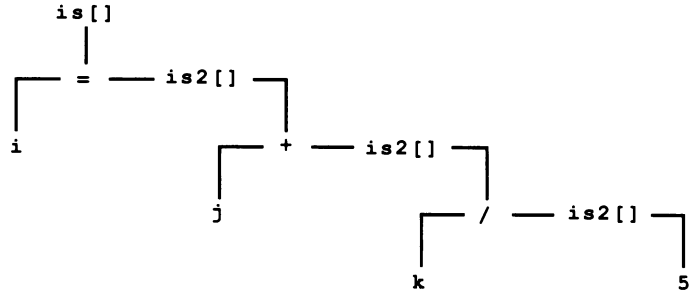


Figure 26-2. Is[] Arrays for i=j+k/5;

So, what kinds of information is carried in these arrays? We shall now see; however, the usefulness of this information will not become clear until the subordinate analyzer functions are studied. Nevertheless, for an introductory look and for future reference, a summary of the contents of the **is[]** arrays is provided in Table 26-1.

Array Element	Contains
is[ST]	address of symbol table entry, else zero
is[TI]	data type of indirectly referenced object, else zero
is[TA]	data type of address, else zero
is[TC]	type of constant, else zero
is[CV]	constant value
is[OP]	p-code of highest binary operator
is[SA]	stage address of "oper 0" code, else zero

Table 26-1. Is[] Array Contents

A brief description of each element follows:

Is[ST] contains the address of the symbol table entry that describes the operand. Its purpose is to allow information from the symbol table to be used in generating code that accesses the operand. Since constants are not represented in the symbol table, however, this element sometimes contains zero. Obviously, this information could have value only with respect to a primary operand. Therefore, when a primary operand combines into some larger entity, this element loses its significance and so is reset to zero.

Is[TI] contains the data type of indirectly referenced objects. These are objects which are referenced by way of an address in a register. This includes (1) function arguments, (2) local objects, and (3) globally declared arrays. In the first two cases, the address is calculated relative to BP, the stack frame pointer. In the third case, an array element's address is calculated relative to the label which identifies the array. Static objects other than arrays, on the other hand, are directly referenced by their labels, so they produce zero in this element. In some cases (e.g., an array name without a subscript or a leading ampersand (&)) the address is all that is desired, and so no action is taken based on this information.

Is[TA] contains the data type of an address (pointer, array, &variable); otherwise, zero.

Is[TC] contains the data type (**INT** or **UINT**) if the (sub)expression yields a constant value; otherwise, zero. The unsigned designation applies only to values over 32767 that are written without a sign (Chapter 3).

Is[CV] contains the value produced by a constant (sub)expression. This element has alternate uses when it is known that the (sub)expression is not a constant.

Is[OP] contains the p-code that generates the highest binary operator in an expression. Its purpose is to determine, in the case of expressions of the form:

```
left oper zero
```

(where **left** is the left subexpression, **oper** is a binary operator, and **zero** is a subexpression that evaluates to zero), which form of optimized code to generate. This element is used by **test()**.

Is[SA] contains the staging buffer address where code that evaluates the **oper zero** part of an expression of the form:

```
left oper zero
```

is stored. If not zero, it tells **test()** that better code can be generated and how much of the end of the staging buffer to replace.

Test() Of the three lead-in functions, **test()** is the most complicated. Its purpose is to generate code to evaluate and test an expression (or expression list), for *true* or *false*. If an expression list is given, the expressions are evaluated from left to right, with the last expression determining the outcome.

This function receives two arguments—a label number for the *false* branch and a Boolean value indicating whether or not to require parentheses around the expression. As with **expression()** above, this function calls **level1()** and so must allocate an initial **is[]** array.

It then calls **need()** to enforce an open parenthesis (if told to). Next it falls into a loop setting the staging buffer, calling **level1()**, and checking for a comma. Each comma means that another expression follows, so it dumps the staging buffer to the output file and continues the loop. After the loop (if told to) it enforces a closing parenthesis.

Finally, before dumping the code for the last expression in the list, **test()** attempts to do some optimizing. If the expression yields a constant (**is[TC]** is *true*) then there is no need for code to evaluate the expression. If the constant value in

is[CV] is *true*, the *false* jump can never be taken, so **test()** simply returns without generating anything; actually, **clearstage()** is called to delete the code that was generated by the last expression. The effect on the program is that control simply goes directly into the portion of the statement that executes under the *true* condition. If the expression is *false*, however, an unconditional jump to the indicated label is generated.

Even though the last expression in the list does not yield a constant, it may still be optimizable. If its form is:

```
left oper zero
```

where **oper** is a relational operator, then instead of generating the normal code for testing the expression:

```
OR AX,AX
JNE $+5
JMP label
```

(p-code **NE10f**), it is possible to take advantage of the fact that the 8086 CPU has several instructions for comparing values to zero. By replacing the code for the:

```
oper zero
```

portion of the expression with these instructions, less and faster code results. So, if **is[SA]** is nonzero and **is[OP]** contains the p-code for one of the relational operators, **zerojump()** is called to replace the unwanted part of the expression with better code. The preferred p-code is passed to **zerojump()** as an argument.

For an example of the improvement this provides, see Listing 19-23. Table 26-2 shows the original p-codes and the ones that replace them. Most of the substitution p-codes in this table have the form:

```
xx10f
```

where ***xx*** designates the type of comparison (e.g., **EQ** for equal), **1** refers to the primary register (AX) which is to be tested, **0** designates the value against which it is to be tested, and **f** means that a jump is generated on the *false* condition.

Notice that **LE12u** maps to **EQ10f** since unsigned values, by definition, cannot be less than zero, making equality with zero the only possibility. Also, **GT12u** maps to **NE10f** for a similar reason. The case of **GE12u** receives the ultimate optimization; no code is generated since unsigned values are always equal to or greater than zero. Finally, the p-code **LT12u** maps to an unconditional jump to the *false* label since unsigned values can never be negative.

ORIGINAL P-CODE	OPTIMIZED P-CODE	TYPE COMPARISON	COMPARISON
EQ12	EQ10f	signed	equal
LE12u	EQ10f	unsigned	less than or equal
NE12	NE10f	signed	not equal
GT12u	NE10f	unsigned	greater than
GT12	GT10f	signed	greater than
GE12	GE10f	signed	greater than or equal
GE12u	--	unsigned	greater than or equal
LT12	LT10f	signed	less than
LT12u	JMPm	unsigned	less than
LE12	LE10f	signed	less than or equal

Table 26-2. Original and Optimized Relational Test P-codes

The Pipeline Functions

Now we look at what, for lack of a better term, I call the *pipeline* functions—the ones through which the **level** functions call each another. Their purpose is to collect in one place logic that would otherwise have been replicated in many **level** functions. It is helpful to view these functions an extension of the **level** functions that call them.

Before going into these four functions, however, we shall consider a few techniques which they have in common. First, **level3()** through **level12()** do not

directly scan for the operators at their precedence levels; they relegate that common task to the underlying instance of **skim()** or **down()**. They do this by passing a string containing a list of operators to the called function. These functions then pass it on to **nextop()** which compares the current token against each operator in the list until it either finds a match or exhausts the list. It returns *true* on success, and *false* on failure. It also sets two global variables for use by the caller. **Opsize** is set to the length of the token, if one is recognized; this is passed by the calling function to **bump()** when it is time to bypass the token. **Opindex** is set to designate which operator in the list matched—zero for the first, one for the second, and so on.

Another common technique is the means by which the pipeline functions know which hierarchy level to call next. This is accomplished by having the calling **level** function pass to the first pipeline function the address of the next **level** function. It simply passes the function name as an argument. (Recall that a naked function name evaluates to the address of the function.) This is passed from one pipeline function to the next until **down1()** finally calls the next level with the statement

```
k = (*level)(is);
```

in which **k** receives the Boolean value indicating whether or not the subexpression parsed by the call is an lvalue, **level** is the address of the target function, and **is** is the address of the array for conveying to higher levels the attributes of the subexpression being parsed.

Skim() This function is only called when the || or **&&** operators are sought—levels 3 and 4, respectively. Essentially this is an embellished version of **down()** that generates additional code to jump over the remainder of a series of the specified operators when the outcome is known. For instance, the outcome of

```
(a > b) || (c == d) || (e != f)
```

is known if

```
(a > b)
```

is *true*. The remaining subexpressions need not—indeed must not—be evaluated. The C language guarantees that these trailing subexpressions will not be executed, and many programs depend on this behavior of the logical operators.

Similarly, the outcome of

```
(a > b) && (c == d) && (e != f)
```

is known if

```
(a > b)
```

is *false*. So the purpose of this function is to *skim* from left to right across a given level of the expression, ensuring that this rule is enforced. **Skim()** falls into a loop in which it first calls **down1()** to parse a subexpression—perhaps only a single primary operand. It then looks to see if the next token is the one being sought (|| or **&&**). If not, it simply returns the value it got from **down1()**. If it is, however, three things occur:

1. the operator is bypassed,
2. if this is the first instance of the operator in the present series, a number for the *dropout* label is allocated, and
3. the function **dropout()** is called to generate the code that tests for the dropout condition. **Dropout()** first, if necessary, generates code to either fetch the subexpression value (an unfetched lvalue), or to load it (a constant) directly with p-code **GETw1n**. It then generates either **EQ10f** (for ||) or **NE10f** (for **&&**) to jump to the dropout label under the right condition. Remember that these codes produce a jump on the *false* condition. The specific p-code that applies is determined by the second argument that **skim()** receives.

Finally, when the series ends:

1. **dropout()** is called for the last subexpression,
2. code is generated to load AX with the outcome (zero or one depending on the fourth argument) when none of the subexpressions indicates otherwise,

3. an exit label number is allocated and a jump to it is generated,
4. the dropout label is generated,
5. code is generated to load AX with the outcome when one of the subexpressions determines it,
6. the exit label is generated,
7. appropriate elements of **is[]** are reset, and
8. zero (meaning that nothing needs to be fetched) is returned.

For an example of the code generated by this sequence, see Listing 19-18.

Down() **Down()** is much simpler than **skim()**. It is called by each level of the hierarchy that looks for binary operators. It first passes control down the *left leg* (by way of **down1()**) to parse an operand that might be on the left side of the binary operator. On receiving control again (the operand has been parsed), if it sees one of the anticipated operators, it passes control down the *right leg* (by way of **down2()** then **down1()**) to parse the right hand operand and to generate code for the operation itself. The first call

```
k = down1(level, is);
```

is simple enough and uses only techniques already described. Also, the examination of the current token to see if it is one of the anticipated operators

```
if(nextop(opstr)==0) return k;
```

is simple. If one of the anticipated operators is not found, **k** is returned to the next higher level.

If **nextop()** returns *true*, however, the operator in the list (**opstr**) indicated by **opindex** has been identified. In this case, **fetch()** is called to generate code to load the left operand (if an unfetched lvalue) into the primary register, and control falls into a loop. The loop bypasses the matched operator then calls **down2()** to evaluate the right operand and generate code for the matched operator. This continues until the operators at the current precedence level (and at the current level of parenthesized nesting) have been exhausted. Finally, zero is returned to

the next higher level, indicating that nothing needs to be fetched (**down2()** fetched the right operand and applied the operator).

Only one thing about this function is not obvious—the way **down2()** is told which operator is being parsed. To understand this, we must recall that global integer arrays **op[]** and **op2[]** contain p-codes for the signed and unsigned binary operators, respectively. Except for seven elements, both arrays are the same. The p-codes are assigned to the array elements according to their level in the parsing hierarchy as indicated in Table 26-3. Within each level they are ordered the same as they appear in the operator lists passed to **down()**. This is important as we will see shortly. Some operators are not represented in these tables because some **level** functions do not call **down()**.

Each time **down()** calls **down2()**, one of the operators in its list (its first argument) has been recognized, and the corresponding entries (p-codes) from these arrays are passed to it. Now, as we saw, **nextop()** sets the global integer **opindex** to designate which operator in the list of operators was found. By design, these lists follow the order in Table 26-3. Finally, notice that **down()** also receives an argument that is the offset into these arrays of the first operator in the list. By adding this offset **opoff** to **opindex**, a subscript into the arrays for the matched operator is created. This is how **down()** determines the p-codes to pass to **down2()**.

One last thing to note about **down()** is that when an operator has been recognized, **down()** declares a new **is** array called **is2[]**. It passes this array along with **is[]** (which it received as its fourth argument) to **down2()**, which must be able to compare the separate attributes of the left and right operands.

As long as **down()** keeps recognizing operators in its list (operators at the same precedence level) it keeps on calling **down2()** in this way. This is what establishes the left-to-right association of the operators that **down()** parses. When no more operators in its list are seen, it returns zero, indicating that no operand fetch is pending.

ELEMENT	OP	OP2	OPERATOR	LEVEL
0	OR12	OR12	|	level5
1	XOR12	XOR12	^	level6
2	AND12	AND12	&	level7
3	EQ12	EQ12	==	level8
4	NE12	NE12	!=	
5	LE12	LE12u	<=	level9
6	GE12	GE12u	>=	
7	LT12	LT12u	<	
8	GT12	GT12u	>	
9	ASR12	ASR12	>>	level10
10	ASL12	ASL12	<<	
11	ADD12	ADD12	+	level11
12	SUB12	SUB12	-	
13	MUL12	MUL12u	*	level12
14	DIV12	DIV12u	/	
15	MOD12	MOD12u	%	

Table 26-3. Contents of the Op[ ] Arrays

Down1() Not much needs to be said about this function. It receives the address of the target **level** function, remembers the current position in the staging buffer, calls the target function, and, if **is[TC]** indicates that a constant value resulted, reverts the staging buffer back to its original position, thereby throwing away the code that was generated. This discarding of the generated code keeps happening with the application of each operator as long as constants result. If an operation does finally produce a non-constant result, then either **skim()** (through **dropout()**) or **down2()** generates one instruction to load the constant before it enters into the operation. If the entire expression yields a constant then **level1()** generates an instruction to load the value into the primary register—the only code produced by the expression.

Down2() **Down2()** is probably the most difficult analyzer function. It cannot possibly make sense without an understanding of the material presented thus far. At this point, though, we should be ready for it.

Remember that this function is called by **down()** when a binary operator has

been recognized. (An exception is the call from **level14()** which we shall not consider here.)

Besides the signed and unsigned p-codes mentioned above (**oper** and **oper2**) and the target level (**level**), **down2()** receives two **is** arrays—**is[]** containing the properties of the left operand that has already been parsed, and **is2[]** that will receive the properties of the right operand. **Is2[]** was declared locally in the preceding instance of **down()** and will be deallocated when it exits. This temporary array is passed to the target function, after which it appears to lower instances of **down2()** as **is[]**. Since **is2[]** is temporary, **down2()** must indicate in **is[]** the properties of the subexpression resulting from this binary operation.

Since this function is so large and obscure, a pseudocode version is presented in Listing 26-1. This listing follows the function exactly, making it easy to read the pseudocode as commentary on the function. Further explanation is given below; before that, however, some of the terms, abbreviations, and conventions used in Listing 26-1 need explaining.

The terms *primary* and *secondary* refer to the primary register (AX) and secondary register (BX), respectively.

The term *constant* (abbreviated *c*) stands for a constant operand that may be the left or right subexpression or the result of the operation.

The term *variable* (abbreviated *v*) stands for a variable operand that may be the left or right subexpression.

The term *address* (abbreviated *a*) stands for an address operand that may be the left or right subexpression. This could be a pointer, an array name, the result of the address operator (**&**), or the result of adding to or subtracting from an address.

The term *int address* (abbreviated *ia*) stands for an integer address that may be the left or right subexpression. This is the address of an integer, as opposed to a character.

The term *left* (abbreviated *l*) stands for the left subexpression. It asserts nothing about its attributes.

The term *right* (abbreviated *r*) stands for the right subexpression. It asserts nothing about its attributes.

The term *both* stands for both left and right subexpressions. It asserts nothing about their attributes.

The term *zero* (abbreviated *z*) stands for the constant value zero.

The term *op* stands for the operator that has been recognized and whose subexpressions are being parsed.

The term *result* stands for the outcome of the operation in question (*l op r*).

The term *pass* refers to the act of passing information up to higher parsing levels by way of side effects in the left is array. Where this occurs, the comment column indicates which element of the array is used.

The plus sign (+) stands for the addition operator.

The minus sign (-) stands for the subtraction operator.

The plus and minus signs together (+-) stand for either an addition or subtraction operator.

The use of parentheses indicates that the symbols within show the form of the subexpression surrounding the operator.

Finally, the use of capitalization refers to the generation of code that performs the indicated actions at run time. Lowercase actions, on the other hand, occur at compile time. The presence of uppercase actions means that the code to accomplish them is being generated (or modified) at the indicated point in the algorithm.

```
   PSEUDOCODE                                        COMMENTS
 1 save stage current address                        before, start
 2 assume result will not be (l op z) or (z op r)    is[SA]=0
 3 if left == constant                               (c op r)
 4     parse right into PRIMARY
 5     if left == zero                               (z op r)
 6         pass current stage address                is[SA]=snext
 7     if right == int address                       double()
 8     and op == add or subtract
 9         double constant                           (2*c +- ia)
10     LOAD CONSTANT INTO SECONDARY                  GETw2n
11 else                                              (v op r)
12     PUSH LEFT ONTO STACK                          PUSH1
13     parse right into PRIMARY                      down1()
14     if right == constant                          (v op c)
15         if right == zero                          (v op z)
16             pass original stage address           is[SA]=start
17         purge PUSH instruction and adjust csp
```

	PSEUDOCODE	COMMENTS
18	if op == add	commutative
19	if left == int address	double()
20	double constant	(ia +- 2*c)
21	LOAD CONSTANT INTO SECONDARY	GETw2n
22	else	not commutative
23	MOVE PRIMARY TO SECONDARY	MOVE21
24	if left == int address	double()
25	and op == add or subtract	
26	double constant	(ia +- 2*c)
27	LOAD CONSTANT INTO PRIMARY	GETw1n
28	else	(v op v)
29	POP LEFT INTO SECONDARY	POP2
30	if left == int address	double()
31	and op == add or subtract	
32	and right == not address	
33	DOUBLE PRIMARY (RIGHT)	(ia +- 2*v)
34	if right == int address	double()
35	and op == add or subtract	
36	and left == not address	
37	DOUBLE SECONDARY (LEFT)	(2*v +- ia)
38	if op == binary	
39	if left or right is unsigned	
40	select unsigned operation	oper=oper2
41	if both constants	(c op c)
42	pass constant designation	is[TC]
43	pass constant value	is[CV]=calc()
44	purge RIGHT CODE	
45	if unsigned result	
46	pass unsigned constant designation	is[TC]=UINT
47	else	
48	pass variable designation	is[TC]
49	OPERATION	gen(oper,0)
50	if op == subtract	
51	and both int addresses	(ia - ia)
52	DIVIDE PRIMARY (RESULT) BY 2	integers between
53	pass operator p-code	is[OP]=oper
54	if op == add or subtract	
55	if both are addresses	(a +- a)
56	pass <result not an address>	is[TA]=0
57	else	
58	if right == address	(? +- a)
59	pass right table address	is[ST]=is2[ST]
60	pass right indirect data type	is[TI]=is2[TI]
61	pass right address data type	is[TA]=is2[TA]
62	else	
63	pass left by default	
64	if left not in symbol table	
65	or right is in symbol table and is unsigned	

```
    PSEUDOCODE                                  COMMENTS
66          pass right symbol table entry       is[ST]=is2[ST]
67      else
68          pass left symbol table entry or zero    default
```

Listing 26-1. Pseudocode Representation of Down2()

Now, with these preliminaries out of the way, we examine Listing 26-1. First, the current stage address is saved so that code generated beyond this point can be purged from the staging buffer. The next line sets **is[SA]** to zero as a default assumption that the subexpression will not have either of the forms *(l op z)* or *(z op r)* that can be optimized within **test()**. If this assumption proves to be wrong it will be changed later.

Next there is a check (line 3) to see if the left operand is a constant. (Recall that the left operand has already been parsed and **is[TC]** indicates whether or not it is a constant.) If so, we have a subexpression of the form *(c op r)* and the code that would have evaluated the constant has already been purged from the staging buffer by **down1()**. So at this point we know that the left side is a constant and we have its value in **is[CV]**, but no code for it exists yet. The important thing here is that, since there is no code yet for the left operand, there is nothing to save on the stack with a push (as in line 12).

In this case, the right operand is parsed (line 4) to generate code which evaluates it. After that the primary register (at run time) contains its value. Now, before generating code for the operator (line 49), if *left* is zero, the current stage address (after *right*, but before *left* and *op* are generated) is passed through **is[SA]** for use by **test()** in its optimizing efforts. Now, if **right** is an integer address and the operation is addition or subtraction, the constant in **is[CV]** is doubled by shifting it left one bit and (doubled or not) code to load it into *secondary* is generated. This (lines 7–11) all occurs in the source statement

```
gen(GETw2n, is[CV] << double(oper, is2, is));
```

The reason for doubling the constant is that it must displace the address by the indicated number of objects (integers), and there are two bytes per integer. After this, control resumes at line 38.

Lines 12–37 similarly look for other cases requiring special handling; beginning with line 12, the subexpression must be some variant of the form (*v op r*). Since *left* is variable, it must be preserved on the stack (line 12) at run time so that the primary register can be used in evaluating *right*. Since *left* must be in the secondary register when the operator is applied, it would be tempting to simply move it from *primary* to *secondary* before parsing *right*. *Right*, however, may require the use of *secondary*, so we cannot safely do that. Therefore, we push it onto the stack, intending to pop it into *secondary* just before the operator is applied. Now *right* is parsed (line 13), after which **is2[]** can be tested to see what it produced.

If it turns out to be a constant (line 14), lines 15–27 are effective. Furthermore, if the constant is zero (line 15), the stage address on entry to this function is passed up through **is[SA]** where, as we saw earlier, **test()** may eventually use it to replace everything generated in this instance of **down2()** with optimized code. Since *left* is not a constant, code for it was already in the staging buffer before entering this function, so **is[SA]** designates the point in the buffer immediately after the calculation of *left*. Therefore, **test()** will not purge that code. Now, since *right* is known to be constant, the push instruction that was just generated is purged from the staging buffer (line 17) since *right* can be loaded directly into the appropriate register at the right time. Furthermore, since *left* is currently (at run time) in *primary* where it was originally calculated, it can simply be left there or moved to *secondary*, depending on where it is needed; no push/pop sequence is needed. The compiler-relative stack pointer, **csp**, is also increased by two to account for the fact that the purged push had decreased it by two.

Now, if the operation at hand is commutative, it doesn't matter which register the operands are in when the operator is applied. Thus, there is no need for code to move *left* to *secondary* where it would ordinarily need to be; it can stay in *primary*, and *right* can be loaded directly into *secondary*. First, however, the constant will need to be doubled if *left* is an integer address and *op* is addition. This is all shown in lines 18–21. The only commutative operation recognized is addition.

On the other hand, if *op* is not commutative (line 22), *right* (a constant) must be loaded into *primary* (line 27); but *left* must first be moved to *secondary* (lines 23). Also before loading the constant, it may have to be doubled (lines 24-26) as before. Finally, code to load *right* into *primary* is generated. From here, control resumes at line 38.

Lines 29-37 show what occurs when both *left* and *right* are variable. First, *left* is popped from the stack into *secondary* where it needs to be when the operator is applied. (Recall that *primary* contains *right* which was just parsed.) Then, if necessary, code to double *left* or *right* is generated. This is because the opposite side is an integer address, the doubled side is not an address, and the operation is either addition or subtraction—conditions that are verified by **double()**. This doubling action differs from that performed on constants, in that it occurs at run time, not compile time. This is done to *right* in line 33 and to *left* in line 37.

The remainder of **down2()** generates the code that actually performs the designated operation, but only if the operator is a binary operator—argument **oper** is not zero. The purpose of this check is to exclude the simple assignment operator (=) from **level1()** and the left bracket and left parentheses from **level14()**. A quick look at these functions will show that they pass zeroes for **oper** and **oper2** in the cases mentioned. The other unary operators (in **level13()**) do not go through this function and so present no problem. The code for these excluded operators is generated in the **level** functions themselves. The distinction between the simple assignment (=) and the other assignment operators (like +=) is because the latter operators are really a combination of two operations—first a binary operation, then an assignment.

First, a choice is made between the signed and unsigned versions of the operation. In most cases, these are the same, but in seven cases they are different (Table 26-3). **Oper2** is the unsigned operator's p-code. If either operand is unsigned, an unsigned operation is to be performed. This is effected by copying **oper2** to **oper** (initially the signed p-code) which is used later to generate code for the operator. The function **nosign()** is called once for each operand, to determine whether or not it is unsigned. It is considered to be unsigned if it is (1) an address, (2) an unsigned integer constant (greater than 32767), or (3) an unsigned variable (integer or character).

If both operands are constants, **calc()** is called to calculate (at compile time) the result and pass it up through **is[TC]** and **is[CV]** (lines 42-43). Then the code generated by *right* is purged (line 44). What happened to the code for *left*, since it too must be purged? The answer is that the lowest instance of **down1()**, through which *left* was parsed, purged the code when it saw that *left* was a constant. So the code for *left* was not even in the staging buffer when **down2()** was entered to evaluate *right*.

At this point, we must make sure that if either constant is unsigned, the result is likewise designated unsigned. **Is[TC]** indicates this already for *left* and, since that is also where the result must be indicated, it follows that we need only inspect **is2[TC]**. If it is found to be unsigned, forcing that designation in **is[TC]** indicates the true result. There are only two possible values for constant types—**INT** and **UINT**. So, if **is2[TC]** contains **UINT** the same value is forced into **is[TC]**.

If either side is variable, then a variable result must be calculated (at run time) by code that applies the pertinent operator to the two registers that are now set up properly. This code is generated (line 49) by the statement:

```
gen(oper, 0);
```

The passing of the constant or variable designation of the result (lines 42 and 48) is accomplished coincidentally with the decision that the result is or is not a constant (line 41). Therefore, lines 42 and 48 do not occur in the source listing.

Now (at run time) *primary* contains the result which must be checked for another possible adjustment (lines 50-52). If the current operation is a subtraction and each operand was an address, then the result is interpreted as the number of objects between them. Furthermore, if the addresses point to integers, then the difference must be divided by two (line 52).

Note that senseless combinations of addresses are possible, such as when one address points to integers and the other to characters, or when the addresses refer to different arrays (makes sense only by presuming a knowledge of how the compiler allocates storage). Small C does not try to deal with these, it only verifies that both addresses point to integers and generates the code to adjust the

result. This covers the case of elements in a single array, and allows for pointers of the same type to be used in ways that make sense. If the data types are mismatched, no adjusting code is generated.

Finally, if addition or subtraction (the only two sensible address operations) is being performed, we must tell the higher parsing levels whether or not the result is an address. If both sides are addresses then the result should not be considered an address, so **is[TA]** is set to zero (line 56). That failing, then one side or the other or perhaps both are not addresses. Here again, since **is[]** (which classifies the result) already has **left** classified, we only need to inspect *right* to see if it should override what is already there. Thus, if *right* is an address, then *left* must not be, and so we have the form (*? +- a*) which means that the result must be classified as an address. In that case, as lines 59-61 show, three attributes from *right* are passed up the hierarchy—**ST**, **TI**, and **TA**. In all other cases, these attributes are passed by default from *left*. This logic makes sure that the forms (*a +- r*) and (*l +- a*) pass the attributes of the address to higher parsing levels.

Note: Adding two addresses yields nonsense. Anything we say about it is wrong, so there is no point in going to the trouble of excluding that case; it would only make the compiler larger and it would still produce a lie. Small C declares that such a result is not an address.

The Precedence Levels

Level1() This function is small but subtle. It recognizes and generates code for the operators at the lowest precedence level, the assignment operators =, |=, ^=, &=, +=, -=, *=, /=, %=, >>=, and <<=. Since these operators group from right to left, the assignments must be made in the reverse order from which the expression is scanned. Therefore, after parsing the left subexpression, if one of these operators is recognized and if the parsed subexpression produced an lvalue, then **level1()** calls itself again. On parsing the second subexpression, the cycle repeats and continues to repeat until the series of assignments ends. At that point, **store()** is called to generate code for the right-most assignment, then control returns to the previous instance of **level1()**. Assignment code is again generated and control again returns to the prior instance. This continues until the initial instance of

level1() returns to one of the lead-in functions or to **primary**() or **level14**() (by way of **down1**() and **down2**()).

As already pointed out, **level1**() first calls **level2**() to parse whatever comes first (or next) in the expression. Since the lower lying functions do not recognize the assignment operators, control returns when an assignment operator (or the end of the expression or something unrecognizable) is reached. The attributes of the parsed subexpression are then found in **is**[]. If **is**[**TC**] indicates that the subexpression yielded a constant, then no code has been generated yet, so **GETw1n** is generated to load the constant into the primary register.

Now tests are made for one of the assignment operators. It may be that none are found; if so, control returns to the caller. If an assignment operator is recognized, however, the local variable **oper** is set to the p-code for the binary operation that is to be performed before assignment. Likewise, **oper2** is set to the unsigned version of the same p-code. This identifies the operation to **down2**(), as we have seen. The simple assignment (=) is a special case because no other operation is implied; in that case, zero is assigned to **oper** and **oper2**. This causes **down2**() to generate no code for a binary operation; it simply returns control back here, where code for the assignment is generated.

If an assignment operator is recognized, a check is made to ensure that the left subexpression is an lvalue, meaning that assignment is possible and legal. That failing, "**must be lvalue**" is issued and control returns.

Assuming the target is an lvalue, two of its attributes, **ST** (address of the symbol table entry) and **TI** (data type if the reference is indirect), are saved in a local array **is3**[] for use in making the assignment. This is because **is**[] will be altered when the right side is evaluated.

Next, two general cases are considered—the case of an indirect reference to the target (by means of an address in the primary register) and the case of a direct reference (by means of a label).

If the target reference is indirect, then the primary register contains its address which must be preserved while the right side is being calculated. This is handled automatically by **down2**() as we have already seen. If a binary operation is being performed, however, the original value of the target must be fetched into

the primary register before the right side is evaluated, and that would destroy the address. So the address is first pushed onto the stack. Then after evaluating the right side and performing the binary operation, the address is popped to the secondary register where the assignment code generated by **store()** expects to find it. Of course, if only a simple assignment (=) is being performed, the push/fetch/pop sequence is not necessary; in that case, **down2()** sees to it that the target address migrates to the secondary register.

Direct references are simpler since, in those cases, the code generated by **store()** makes use of a label to locate the target. If a binary operation applies, as before, the original value of the target must be fetched; however, there is no need to protect a target address as before. So **down2()** is simply called to generate code that evaluates the right side and applies the operation in question. On the other hand, if the operation is a simple assignment (=) there is no need to fetch the initial value of the target or to go through **down2()**, so **level1()** calls itself directly to parse the right side.

Finally, before returning, **level1()** calls **store()** to generate code that stores the primary register in memory at the target address. It passes **is3[]** which contains information **store()** needs about the target.

Level2() This level parses the conditional operator (**?:**). Recall that this operator has the form:

```
Expression1 ? Expression2 : Expression3
```

The operation of **level2()** is quite straightforward. First, it calls **level3()** by way of **down1()** to parse the first expression. Then, if the next token is not a question mark, we do not have a conditional operator, so control returns to the caller. By calling **level3()**, assignment and conditional operators are disallowed in the first expression—that is, unless it is enclosed in parentheses.

On the other hand, if this is a conditional operation, then **getlabel()** is called to reserve a label number for use in jumping around ***Expression2***, and **dropout()** is called to generate code to perform that jump if ***Expression1*** is *false*.

Next, ***Expression2*** is parsed by recursively calling **level2()** through

down1(). Notice that since **level2**() is called, ***Expression2*** may contain the conditional operator, but not assignment operators. Of course, with parentheses, it can include any operator.

Now, if necessary, **fetch**() obtains the expression's value from memory, or the expression's constant value is loaded directly.

Next, the second character of the operator (the colon) is enforced by:

```
need(":");
```

which complains on not finding it.

Having parsed ***Expression2***, it is necessary now to generate an unconditional jump around ***Expression3*** so that it too will not be executed when ***Expression2*** is selected. As before, **getlabel**() reserves an exit label number. This is then passed to **gen**() for use in generating the jump.

Now, before parsing ***Expression3***, we generate the target label for the *false* jump that we generated earlier by calling **dropout**(). ***Expression3*** is then parsed just as ***Expression2*** was. Last of all, the exit label is generated.

This completes the processing of the conditional operator, but it is still necessary to decide what attributes to pass up to the calling level. Three arrays—**is1**[], **is2**[], and **is3**[]—are used in parsing the three expressions respectively. **Is1**[] is received from the caller, so it must convey the attributes of the result back up the line.

The task of determining the correct attributes is complicated by the fact that either of two separate expressions is selected at run time, but we must decide at compile time how to treat the result in the further process of expression evaluation. Because of this, the two expressions must either have similar attributes, or it must be clear which expression's attributes to choose if they differ.

First, if both expressions (2 and 3) yield constants, the compiler assumes that they have different values, since otherwise there would be no sense in writing the conditional operator. Therefore, since a choice is being made at run time between two different constants, the result is designated as variable. Furthermore, the following properties are asserted: it is not an address, nothing is to be fetched indirectly, and no symbol table entry applies. These are all accomplished by zeroing the **TC**, **TA**, **TI**, and **SA** entries of **is1**[].

The next two acceptable cases are where either expression yields a constant. In these cases, the result is given the attributes of the non-constant expression. This is based on the assumption that the constant is a special value of whatever the non-constant expression would otherwise have computed. Typically, these might be an address expression and the constant zero—the null address.

The last acceptable possibilities are that both expressions yield either addresses or non-addresses. In these cases, the choice of attributes is arbitrary since both expressions agree—the attributes of the third expression are chosen.

Should all four cases fail, the message "**mismatched expressions**" is issued.

Level3() Through Level12() After the material already covered, this should be easy. These ten functions are essentially alike and very simple. Each one descends to the next lower level in the parsing hierarchy by way of either **skim()** or **down()**. In so doing, it passes a string of the operators to be recognized at the current level, the address of the target **level** function, the address of the **is[]** array that it received, and various other arguments.

Basically, these function only serve to direct the flow of control through the central part of the expression analyzer. In so doing, they enforce the operator precedence rules. We have already seen the function that calls them (**down1()**) and the ones that they call (**skim()** and **down()**).

Level13() This function handles the unary operators ++, —, ~, !, -, *, &, and **sizeof()**. Listing 26-2 is a pseudocode version of this function.

```
   PSEUDOCODE                                  COMMENTS
 1 if op == pre-increment                      ++
 2     parse operand recursively               level13()
 3     if operand is not an lvalue
 4          issue "must be an lvalue"
 5          return <request no fetch>
 6     generate rINC1 on operand in memory     step()
 7     return <request no fetch>
 8 if op == pre-decrement                      -
 9     parse operand recursively               level13()
10     if operand is not an lvalue
11          issue "must be an lvalue"
12          return <request no fetch>
```

```
    PSEUDOCODE                                          COMMENTS
13      generate rDEC1 on operand in memory             step()
14      return <request no fetch>
15  if op == one's complement                           ~
16      parse operand recursively                       level13()
17      if operand is an lvalue
18          FETCH OPERAND INTO PRIMARY
19      generate COM1
20      pass ~is[CV]
21      pass zero stage address                         is[SA]
22      return <request no fetch>
23  if op == logical not                                !
24      parse operand recursively                       level13()
25      if operand is an lvalue
26          FETCH OPERAND INTO PRIMARY
27      generate LNEG1
28      pass !is[CV]
29      pass zero stage address                         is[SA]
30      return <request no fetch>
31  if op == unary minus                                -
32      parse operand recursively                       level13()
33      if operand is an lvalue
34          FETCH OPERAND INTO PRIMARY
35      generate ANEG1
36      pass -is[CV]
37      pass zero stage address                         is[SA]
38      return <request no fetch>
39  if op == indirection                                *
40      parse operand recursively                       level13()
41      if operand is an lvalue
42          FETCH OPERAND INTO PRIMARY
43      if operand is symbolic
44          pass indirect data type from symbol table   is[TI]
45      else pass indirect data type integer            is[TI]
46      pass zero stage address                         is[SA]
47      pass <not address>                              is[TA]
48      pass <not constant>                             is[TC]
49      pass <do not fetch if function call>            is[CV]
50      return <request fetch>
51  if op == sizeof                                     sizeof()
52      bypass and remember (
53      default size to 0
54      if "unsigned" set size to 2
55      if "int" set size to 2
56      else if "char" set size to 1
57      if size != 0 and "char *" set size to 2
58      if size == 0 and symbol and in symbol table
59          fetch size from symbol table
60      else if size = 0 issue "must be object or type"
```

```
    PSEUDOCODE                                          COMMENTS
61      bypass ) if there was a (
62      pass <integer constant>                         is[TC]
63      pass size as <constant value>                   is[CV]
64      pass <not address>                              is[TA]
65      pass <not indirect fetch>                       is[TI]
66      pass <not in symbol table>                      is[ST]
67      return <request no fetch>
68  if op == address                                    &
69      parse operand recursively                       level13()
70      if operand is not an lvalue
71          issue "illegal address"
72          return <request no fetch>
73      pass address data type from symbol table        is[TA]
74      if indirect object reference
75          return <request no fetch>
76      generate POINT1m
77      pass indirect data type from symbol table       is[TI]
78      return <request no fetch>
79  parse operand at higher precedence level            level14()
80  if op == post increment                             ++
81      if operand is not an lvalue
82          issue "must be an lvalue"
83          return <request no fetch>
84      generate rINC1                                  step()
85      generate rDEC1
86      return <request no fetch>
87  if op == post decrement                             -
88      if operand is not an lvalue
89          issue "must be an lvalue"
90          return <request no fetch>
91      generate rDEC1                                  step()
92      generate rINC1
93      return <request no fetch>
94  return <fetch request status from below>
```

Listing 26-2. Pseudocode Representation of Level13()

Although this is a large function, it is quite straightforward and involves a good bit of repetition. It first attempts to recognize the operators that precede an operand (lines 1, 8, 15, 23, 31, 39, 51, and 68).

++...

If an increment operator is found (line 1), the following operand is parsed (line 2) by calling **level13()** again. This recursive call allows for the fact that

these operands group from right to left, and further occurrences of them (and higher precedence operators) may appear within the subexpression that becomes the operand for the current operator. The resulting operand must be an lvalue; otherwise, it cannot be incremented. If it is not (line 3), the message "**must be an lvalue**" is issued and control returns. On the other hand, if it is an lvalue, code is generated to increment it in memory (line 6). This action also leaves a copy of it in the primary register for use at the current point in the expression. On return, the caller is told (line 7) that there is no need to fetch anything.

The function **step()** is used to generate code for increment and decrement operators. It accepts, as its first argument, one of the p-codes **rINC1** or **rDEC1**. The second argument is the address of **is[]** for the lvalue being stepped, and the third argument is either zero or a second p-code—either **rDEC1** or **rINC1**. First, **step()** calls **fetch()**, passing it **is[]**, to obtain the original value of the object to be stepped. Next, the first p-code is generated to either increment or decrement the object in the primary register. Then **store()** is called to generate code for storing it back in memory. It is important to realize that this leaves the adjusted value in the primary register. If **step()** is called, as here, by a prefix operator, the third argument is zero. This means that there is no need to restore the original value of the object in the primary register. In that case, **step()** simply returns. If the last argument is not zero, however, it is taken as the p-code for a follow-up operation that is generated before returning.

`-...`

The decrement operation is accomplished by equivalent logic (lines 8–14).

`~...`

If the operation is a one's complement, the operand is first parsed by calling **level13()** recursively (line 16). If it yields an lvalue (line 17), code is generated to fetch it into the primary register (line 18). Next, code is generated to perform the one's complement (line 19). Then, if the operand is a constant, its one's complement is performed at compile time in **is[CV]** (line 20), through which the constant value is passed up the line. Since this is a unary operation, the subexpres-

sion acted on by the operator cannot have the form (*left op zero*) or (*zero op right*), so **is[SA]** is set to zero (line 21), preventing **test()**—if it is the active lead-in function—from trying to optimize the code in the staging buffer. Finally, zero is returned to the caller (line 22), indicating that there is no need to fetch anything.

`!...` and `-...`

The logical NOT and unary minus operators receive analogous treatment in lines 23-30 and 31-38, respectively.

`*...`

The indirection operator can be deceiving because the only code it generates is a fetch (line 42) of what is supposed to be an address, if the operand is an lvalue (line 41). The real work here is in (1) declaring the operand to be the address for an indirect reference (lines 43-45), and (2) asserting that it points to an lvalue (line 50). The latter action guarantees that, at a higher point in the analysis, the operand will be fetched (or possibly accepted as the target for an assignment) and the former action guarantees that it will be referenced indirectly. The data type placed in **is[TI]** comes from the symbol table if the operand is based on a symbol; otherwise, the type is forced to integer. Finally, before returning, attributes are passed through **is[]**, indicating that the result (after the anticipated fetch) is not an address, is not a constant, is not to be fetched if the reference turns out to be a function call (the function address is already in the primary register), and does not have the form (*left op zero*) or (*zero op right*).

`sizeof(...)`

The **sizeof** operator must inspect the data type or symbol on its right and return a constant that is the size of the type data specified or the named object. First (lines 54-57) it looks for one of the data type specifications **unsigned**, **unsigned int**, **unsigned int ***, **unsigned char**, **unsigned char ***, **int**, **int ***, **char**, or **char ***. That failing, it tries to interpret the current token as a name in the symbol table (line 58). In the first case, it sets the size directly. In the second

case, it takes the size from the symbol table where it was placed when the object was declared. If both cases fail (line 60), then "**must be object or type**" is issued and zero is returned. Before returning (lines 62-66), it designates the result to be a constant integer and provides its value in **is[CV]**. Other properties are: not an address, no indirect fetch is pending, and no symbol table reference exists for the returned value. Finally, it returns to the caller, indicating that nothing is to be fetched.

&...

The address operator is handled by first parsing the operand (line 68). The operand must be an lvalue (lines 70-72), since things that do not occupy space in memory (like constants) have no addresses. Next, the data type is copied from the symbol table into **is[TA]** (line 73), indicating that the result is an address referring to an object of the specified type. Then, if the object (without the address operator) would have been referenced indirectly, its address is already in the primary register, so there is nothing more to do but return to the caller, asserting that there should not be an attempt to fetch the object. If the object would have been referenced directly, however, then its address is loaded into the primary register (line 76), the result is declared to be an indirect reference by copying its data type from the symbol table to **is[TI]**, and control returns to the caller, asserting that nothing is to be fetched.

At this point (line 79), since the previous tests have failed, none of the operators at this level have been recognized, but it is possible that a post increment or decrement exists, so the operand is first parsed by calling **level14()**. This call parses only a primary object (possibly subscripted and/or a function call) or a subexpression in parentheses. After that, attempts are made to recognize post increment and decrement operations.

...++ and ...—

When these operators are recognized (lines 80 and 87), code to place the operand in the primary register has already been generated so it is only necessary to generate code for the operation in question (lines 84 and 91) and to return the primary register to its original value (lines 85 and 92). As before, zero is returned

indicating to the caller that nothing needs to be fetched.

Finally, if none of the operators at this level are recognized, control returns to the caller with whatever fetch request was produced by the parsing of the current subexpression (line 94).

Level14() **Level14()** is the last (highest) precedence level before **primary()**. **Level14()** recognizes only two operations, subscripting and calling functions. It also handles the case where a function's address is invoked by naming the function without a left parenthesis following. As before, **level14()** is presented in pseudocode in Listing 26-3.

```
    PSEUDOCODE                                  COMMENTS
 1 parse primary operand or subexpression       primary()
 2 advance to next token
 3 if token = [ or (
 4     loop:
 5         if token = [                         subscript
 6             if operand not in symbol table
 7                 issue "can't subscript"
 8                 bypass subscript expression  skip()
 9                 enforce ] token
10                 return <request no fetch>
11             if operand is an address
12                 FETCH INTO PRIMARY if necessary
13             else
14                 issue "can't subscript"
15                 set <request no fetch> for return  k=0
16             save staging buffer address      setstage()
17             set <not constant> in temporary array  is2[TC]=0
18             parse subscript expression       level1()
19             enforce ] token
20             if subscript is a constant       ...[constant]
21                 purge subscript code         clearstage()
22                 if subscript is not zero
23                     if operand is an integer
24                         double constant
25                         generate GETw2n
26                     else generate GETw2n
27                     generate ADD12
28             else if operand is an integer    ...[variable]
29                     generate DBL1
30                  generate ADD12
31             pass <not an address>            is[TA]=0
```

```
    PSEUDOCODE                                    COMMENTS
32          pass indirect type from symbol table    is[TI]
33          set <request fetch> for return          k=1
34      else
35          if token == (                           function call
36              if not symbol table reference
37                  INDIRECT CALL                   callfunc(0)
38              else if not a function reference
39                      if <request fetch>
40                      and not already fetched     is[CV]==0
41                          FETCH OPERAND           fetch()
42                      INDIRECT CALL               callfunc(0)
43                  else DIRECT CALL                callfunc(ptr)
44              pass <not symbol table reference>   is[ST]=0
45              pass <not constant>                 is[TC]=0
46              pass <reference fetched>            is[CV]=0
47              set <request no fetch> for return   k=0
48          else return <pre-set fetch request>
49    loop back
50 if operand is in symbol table and is a function    no (
51    generate POINTm
52    pass <not symbol table reference>               is[ST]=0
53    return <request no fetch>
54 return <fetch request from below>
```

Listing 26-3. Pseudocode Representation of Level14()

First, since the pertinent operators (left bracket and left parenthesis) trail, the operand (address expression or function name) is parsed first. This is done by calling **primary()** (line 1).

If the following token is not a left bracket or left a parenthesis (line 3), control drops down to line 50 where the parsed operand is checked for being a function name. If so, then code to load its address is generated (line 51), the symbol table address in **is[ST]** is zeroed since the reference has already been made, and control returns to the caller with an indication that the operand has already been fetched. But if the operand is not a function name, then nothing needs to be done at this level, so control returns to the caller with whatever fetch request **primary()** returned.

That leaves the cases where either subscripting or function calling is to be done. In these cases, control falls into a loop in which the left bracket and left parenthesis are recognized separately. The loop repeats until neither is seen, at

which point (line 48) control returns with the fetch request from **primary**() or possibly a pre-set fetch request (lines 15, 33, and 47). By looping, the analyzer allows subscripting to be performed as a part of the calculation of a function's address.

The remaining logic breaks down into two parts—subscript analysis (lines 6-33) and analysis of function calls (lines 35-47).

The subscript operator (line 5) must follow a pointer, an array name, or an expression based on either and yielding an address. If there is no symbol table reference for the operand (line 6) it cannot be based on a pointer or array name, so the message "**can't subscript**" (line 7) is issued, the subscript expression is passed over (line 8), the closing bracket is enforced (line 9), and control is returned to the caller (line 10). That failing, if the operand is a pointer (line 11), code to fetch its content is generated (line 12). But if it is not an address, it cannot be subscripted, so "**can't subscript**" is issued and the fetch request is pre-set to zero.

Now it is time to parse the expression comprising the subscript. In case it turns out to be a constant, the current staging buffer address is saved (line 16) so the generated code can be discarded (line 21). Then, since a new array **is2**[] will be receiving the attributes of the subscript expression, element **TC** is initialized to zero (line 17). This will change if the expression does in fact yield a constant. Finally, the expression is parsed by calling **level1**() by way of **down2**() (line 18) and the closing bracket is enforced (line 19).

Now, if the subscript is in fact a constant expression (line 20), the code generated for it in the staging buffer is purged by calling **clearstage**() (line 21). Furthermore, if the constant subscript has the value zero (line 22), no offset to the base address needs to be made, so no code is generated. A non-zero value must produce an offset, however. If the address refers to integers (line 23) then the constant is doubled (line 25) before being loaded into the secondary register. If characters are being referenced, no doubling is needed so the unaltered constant is loaded. Finally, the offset (in the secondary register) is added to the address (in the primary register) to produce the effective address.

On the other hand, if the subscript yields a variable quantity, the code that

evaluates the subscript expression is retained. If integers are being referenced (line 28), the doubling takes the form of code that doubles the subscript at run time (line 29). And, as before, code that adds the base address to the offset is generated (line 30).

Now, all that remains is to set up the side effects in **is[]** and return. **Is[TA]** is zeroed (line 31) to indicate that the object being referenced is not an address, and the data type is copied from the symbol table to **is[TI]** (line 32) indicating that the reference will be indirect through the effective address in the primary register. And, finally, before control loops back to look for other operators at the same level, the return value is pre-set to request that the referenced object be fetched (line 33).

Most of the work in parsing function calls is performed in **callfunc()**, which is described next. In this function, however, the processing is simple. If the operand that designates the function is not a name in the symbol table (line 36), then it must be an expression that produces the function's address. In that case, **callfunc()** is called (line 37) with a zero argument instead of the address of a symbol table entry. This results in an indirect function call. If the operand is in the symbol table but is not identified as a function (line 38), however, it is assumed to be an expression involving variables, arrays, and/or pointers. So, if there is a pending fetch request (lines 39-40), the operand's value is fetched (line 41) before the indirect call is generated (line 42).

Note that the indirection operator in

```
(*func)();
```

is recognized in **level13()** where the fetch operation is generated. At that point **is[CV]** is set to one, indicating to **level14()** that the fetch is satisfied. This special use of **is[CV]** is permissible since the indirection operator precludes the possibility of having a constant subexpression, so **is[CV]** is free to be used in this way. A means of telling **level14()** whether or not to fetch the function address is needed because the indirection operator always returns *true* for a fetch request. If it seems strange that **level13()** is passing something *up* to **level14()**, then notice that the indirection operator is preceded with a left parenthesis, so

this instance of **level13()** is reached by way of **primary()** which in turn was called by **level14()**.

If the operand is a function name in the symbol table, however, **callfunc()** is called with the address of the symbol table entry as an argument, thus causing a direct call to be generated (line 43).

Now side effects are set up; specifically, **is[ST]** is zeroed (line 44) to indicate that the value of the function being called has nothing to do with the symbol table, **is[TC]** is zeroed (line 45) to indicate that the value of the function is variable, **is[CV]** is zeroed (line 46) to indicate that the function address has already been fetched, and the return value is preset to prevent an automatic fetch at a higher level.

Last of all, control loops back to look for other operators at the same precedence level.

Callfunc() As we have just seen, **callfunc()** is called at three points in **level14()**. If its argument is zero, it generates an indirect call through the address in the primary register. But if it is not zero, it specifies the address of the symbol table entry for the function being called, and a direct call to the label bearing the function's name is generated.

This function must generate code for four tasks: (1) to evaluate each argument and push it onto the stack, (2) to provide an argument count to the called function, (3) to call the target function, and (4) to deallocate the arguments from the stack on return.

On entry, the opening parenthesis for the argument list has been recognized and passed over, so the first argument (if there is one) is next. The arguments are processed in a loop that terminates when the right parenthesis (or end of the statement or of the input) is reached. With each iteration, one argument expression is evaluated and pushed.

If an indirect call is being made, then the function address must be preserved as each argument is evaluated. This is done by placing it on the stack temporarily. Then, after evaluation, the argument swaps places with the function address.

The steps are:

1. push the function address on the stack (**PUSH1**)
2. evaluate the next argument expression
3. swap the function address with the argument (**SWAP1s**)

Expression() is the lead-in function for argument evaluation. Notice that the entire expression analyzer is being called recursively through this function. Each actual argument is an expression in its own right, while the function call, of which it is a part, represents part or all of another expression.

When the arguments have been exhausted, a right parenthesis is enforced. Then, if the target function is not **ccargc**(), **ARGCNTn** is generated to load a count of the arguments into the CL register. (Since **ccargc**() is the function that fetches the count, it would not be a good idea to destroy CL with a count of the arguments passed to it.)

Next, if this is a direct call, **CALLm** is generated to call the target label. But if it is indirect, **CALL1** is generated to call the function pointed to by the primary register.

Finally, **ADDSP** is generated to restore the stack to its original value before the arguments were pushed onto it.

Primary Operands

Primary() At the bottom end of the expression analysis hierarchy is **primary**() which looks for the simplest possible operand reference. This may be a variable name, an array name, a pointer name, a character constant, a string constant, or a subexpression in parentheses.

First, **primary**() looks for a left parenthesis and, finding one, calls **level1**() to evaluate the enclosed subexpression, enforces the closing parenthesis, and returns the fetch request from **level1**().

That failing, the entire **is**[] array is set to zeroes, thereby establishing the default values for its elements. Further action is only necessary when zero is not

appropriate.

At this point, the current token must be a symbol or a constant. So **symname()** is called to determine if it is a symbol. If so, **symname()** places it in the local array **sname[]** and passes over it in the input line. Three possibilities exist for a legal symbol: it may be a local reference, a global reference, or a reference to an undeclared function. First, it is sought in the local symbol table; if that fails, the global table is searched; and, if that fails, it is assumed to be an undeclared function. Searching for locals before globals is critical, since local names must supersede global ones. Also, recall that the local table is searched sequentially backward, so that "newer" locals (the ones that are nested deeper) mask "older" ones. If both searches fail, the symbol is added to the global table as a function. Then, if it never is formally declared, it will be identified to the assembler as an external reference before the compiler quits.

On finding the symbol in the local table, a check is made to verify that it is not a label. If it is, the message "**invalid expression**" is issued since labels cannot be referenced in expressions. Control then returns indicating no need for an operand fetch. That failing, we have a proper reference to a local object, so **POINT1s** is generated, which yields:

```
LEA AX,n[BP]
```

to calculate in the primary register the address of the object in question. The letter ***n*** stands for the object's distance from the stack frame address in BP, and comes from the symbol table. It is negative for local objects and positive for arguments.

Next, **is[ST]** is set to the address of the symbol table entry and **is[TI]** is set to the data type, so that the reference will be seen as an indirect one.

Then, if the symbol names an array, **is[TA]** is set to the data type, indicating that the operand is an address, and control returns, requesting no fetch operation. These actions are appropriate since an unsubscripted array name is supposed to produce the address of the array. Of course, subscripting changes this.

If the symbol names a pointer, however, **is[TI]** is changed to indicate an unsigned integer operand. This is done because the data type in the symbol table refers to the type of objects pointed to, whereas **is[TI]** refers to the object being

referenced indirectly—the pointer. True, a pointer is not an integer, but they are the same size and that is all that matters here. As before, **is[TA]** is set to the data type, indicating that the operand (when it is fetched) yields an address.

Finally, whether the symbol is a pointer or a variable, control returns, requesting a fetch operation since the primary register contains the address of the object not its value.

Global references are treated somewhat differently since globals are referenced by their labels and because functions must be handled.

First, **is[ST]** is set to the address of the symbol table entry for the recognized symbol.

Next, if the symbol is an array name, three actions occur. **POINT1m** is generated, which yields:

```
MOV AX,OFFSET _array
```

for loading the array's address into the primary register. ***Array*** stands for the name of the array. Then, since any further reference will be indirect, **is[TI]** is set to the data type of the symbol. And, finally, **is[TA]** is likewise set to the data type, indicating that an address is in the primary register. Control then returns with no request to fetch an operand.

If the symbol is a pointer name, however, only **is[TA]** is set to the data type of the symbol.

Then, whether the symbol is a pointer or a variable, control is returned with a request to fetch the operand. Since **is[TI]** remains zero, the requested fetch will be direct.

If the symbol is not found in the global symbol table, **addsym()** is called to create an entry for it as a function. At this point the symbol is known to be a function and nothing remains to be done but return requesting no fetch operation. **Level14()** will take care of calling the function or loading its address.

Finally, if the current token is not a legal symbol, then it must be a constant or else the expression is illegal. So **constant()** is called to parse it and tell whether or not it was a constant. Last of all, control returns, requesting no fetch operation. Recall that constants are passed back up the line in **is[CV]** with

is[TC] indicating the type of the constant (**INT** or **UINT**).

Table 26-4 displays in a matrix the values that **primary()** returns to higher-level functions. The first six columns correspond to the significant elements of the **is[]** array. The last column shows the value actually returned by **primary()**; **no** refers to zero and **yes** refers to one. The symbol **->st** stands for a pointer to the recognized object's entry in the symbol table. The word **type** refers to the data type of the object, or to which it refers. **UINT** and **INT** are the unsigned and signed integer data types as defined in **CC.H**. And, the unusual looking **(U)INT** refers to either **UINT** or **INT** depending on the value of the constant and whether or not it was written with a minus sign (see Chapter 3). This table should be of considerable value when studying the logic of the higher-level analyzer functions.

		is[ST]	is[TI]	is[TA]	is[TC]	is[CV]	fetch?
LOCAL	array	->st	type	type			no
	pointer	->st	UINT	type			yes
	variable	->st	type				yes
GLOBAL	array	->st	type	type			no
	pointer	->st		type			yes
	variable	->st					yes
	function	->st					no
CONST	numeric				(U)INT	value	no
	character				INT	value	no
	string						no

Table 26-4. Values Returned by Primary()

Constant() This function is called by **primary()** when it thinks the current token must be a numeric, character, or string constant. If so, it parses the constant, generates code, and returns *true*, indicating that a constant was in fact parsed. On failure, it returns *false*.

Corresponding to the three valid possibilities, **constant()** calls **number()**, **chrcon()**, and then **string()** looking for a numeric constant, a character constant, or a string constant, respectively. Each of these functions returns a non-zero value on success, and zero on failure. If all three fail, *false* is returned to **primary()**. These functions generate no code, they just parse the constant and place the result in either **is[CV]** (number or character constant) or the literal pool (string constant). **Constant()** then generates the code to reference the result.

The functions **number()** and **chrcon()** are essentially alike in their effects, whereas **string()** is distinct because it yields an offset to the string in the literal pool rather than a constant value.

Number() and **chrcon()** return the type of the constant found on success—that is, **INT** or **UINT**. As we can see from the listing, the return value in these cases is assigned to **is[TC]** as the type of the constant, or as zero. This informs the higher parsing levels that a constant was or was not found and, if so, its type. They also receive as their only argument the address of **is[CV]** where they place the binary integer form of the constant they find.

Now, if **number()** or **chrcon()** indicates success, **GETw1n** is generated to load the value into the primary register. This produces:

```
XOR AX,AX
```

or,

```
MOV AX,n
```

depending on whether or not the constant is zero.

String constants are different because they yield an address to the string. Where will the string be placed in the program? Recall from the discussion of the literal pool (Chapter 20) that strings are kept there until the end of the function, at which point the pool is dumped into the data segment. Each compiled function allocates a compiler-created label for the dumped pool, and each string is referenced as an offset from that label.

String() receives the address of a local integer **offset** as an argument that it uses as the target for the string's offset in the literal pool. When **string()** indi-

cates success, it has put the string in the literal pool (in the compiler, not in the program) and the offset in **offset**. **Constant**() then generates **POINT1l** which produces

```
MOV AX,OFFSET _m+n
```

where ***m*** is the number of the label for the current function's literal pool and ***n*** is the offset from the label to the first character of the string. The assembler produces from this the *data segment relative* address of the string, and the linker *fixes* it to an absolute address when all of the modules making up the program are finally joined.

Since a string address is not a constant (its actual value is not known at compile time), **is[TC]** and **is[CV]** are left at zero, as they were initialized by **primary**().

To summarize, **constant**() sees to it that a numeric or character constant is placed in **is[CV]** and is loaded (at run time) into the primary register. It ensures that a string constant is copied into the literal pool, and its address is loaded into the primary register at run time.

Number() This function looks for a (possibly signed) decimal, hexadecimal, or octal string and converts it to a signed binary integer.

First it looks for a leading sign. If a hyphen is seen, **minus** is set *true*, otherwise it defaults to *false*. A plus sign is simply ignored, leaving **minus** at its default value of *false*.

Then, if the next character is not a digit, *false* is returned. That failing, the first digit is inspected to see if it is **'0'**—the lead-in for octal and hexadecimal numbers. If not, then each digit found (going from left to right) is converted to its binary equivalent and is added to an accumulator **k**, which is first multiplied by 10. When the digits are exhausted, **k** contains the absolute value of the number. No account is taken of overflow, it simply yields a corrupt value.

Before discussing the application of the sign and what is returned to the caller, let's see what happens when the leading character is **'0'**. In that case, the second character is inspected to see if it is an **x** (uppercase or lowercase). That being true, a hexadecimal number follows, so the **x** is bypassed and a loop is

entered that lasts as long as legal hexadecimal digits are found. In each instance, the digit is converted to its binary value and added to **k**, after multiplying **k** by 16. Since the alphabetic digits do not immediately follow the numeric digits in the ASCII code set, separate statements are required for each case. The result of this loop is the absolute value in binary. As before, no account is taken of overflow.

If there is no **x** following the leading zero, then a different loop is entered that lasts as long as legal octal digits are found. It similarly computes the absolute value in **k**.

After the absolute value has been computed, if **minus** is *true*, the negated value of **k** is stored at the location indicated by the argument, and **INT** is returned. If **minus** is not *true*, however, then **k** is stored without negation. In that case, its value is tested to see if it should be designated as signed or unsigned. This version of Small C treats constants that do not have a minus sign, and are larger than 32767, as unsigned quantities (Chapter 3). If that condition holds, **UINT** is returned, otherwise **INT**.

Chrcon() This function looks for a character constant of one or two characters enclosed by apostrophes. First, **chrcon**() looks for an apostrophe; if that fails, it returns *false*, indicating failure.

Following the leading apostrophe, control falls into a loop that exits when the trailing apostrophe is found. With each iteration an accumulator **k** is shifted left eight bits and the current character is added to the result. The characters are obtained through a filter function **litchar**() which recognizes and evaluates escape sequences; so it is possible that two or more characters may reduce to a single character as seen by **chrcon**(). Since the loop continues until the trailing apostrophe, if more than two characters are present, the leading characters will be lost and only the last two will contribute to the constant.

At the end, the closing apostrophe is bypassed, **k** is placed into **is**[CV], and **INT** is returned.

String() This function parses quoted strings. First, it looks for the leading quote. If that fails, it returns *false*. After seeing the quote, it copies the current value of

litptr (the offset to the current end of the literal pool where the new string is about to be copied) to the location that it received as an argument. It then falls into a loop that exits when the trailing quote is reached. With each iteration, it filters the next character(s) through **litchar**() and passes it on to **stowlit**() for storage in the literal pool. Finally, the closing quote is bypassed, a terminating zero is put at the end of the string in the literal pool, and *true* is returned.

Stowlit() **Stowlit**() places character or integer values in the literal pool. It receives as arguments the value and its size in bytes. If an overflow would occur, it issues "**literal queue overflow**" and aborts the run. If there is room in the literal pool, however, it places the value (one or two bytes) in the pool at the offset indicated by **litptr**. It then increments **litptr** by the size of the value and returns to the caller.

Litchar() **Litchar**() returns the current character in the input line after advancing to the next one. If the current character is a backslash, it is taken as the start of an escape sequence. The backslash is bypassed and the following character is tested to see if it is one of **n**, **t**, **b**, or **f**. If so, that too is passed over and a newline, tab, backspace, or formfeed, respectively, is returned.

That failing, **litchar**() looks for an octal number to convert to binary. Up to three digits will be recognized. Each succeeding character is tested for being an octal digit. If so, it is converted to binary and added to an accumulator **oct** which is first shifted left three bits. The loop ceases when no more octal digits remain or three digits have been processed. Finally, **oct** is returned.

If no octal digits are seen, however, then the character following the backslash is returned as is. This is the case of an escape followed by an unspecified character. Thus, for instance, a sequence of two backslashes would return the second backslash.

CHAPTER 27

Optimizing

We now come to the most refined part of the compiler—its peephole optimizer. The original compiler (by Ron Cain in 1980) did not incorporate optimizing. When I upgraded it to version 2.0 (under NorthStar DOS, then CP/M) I added the staging buffer and a rudimentary peephole optimizer. At that time, the buffer held ASCII assembly code (p-codes were not used) and **streq()** was used repeatedly in a long string of **if...else if...** statements to search for cases to optimize.

Later, when Russ Nelson at Clarkson College ported that compiler to MS-DOS, he introduced p-codes and made appropriate changes throughout the compiler, including the optimizer. The result was more efficient optimizer testing, but the original **if...else if...** structure remained. Furthermore, since every case was unique and the optimizations directly altered the staging buffer, the code was very obscure.

In my own MS-DOS implementation, based on Nelson's work, I replaced that structure with a large **switch** statement to reduce the compiler's size and make the testing faster still.

Finally, with the release of version 2.2, I completely rewrote the optimizer. The result is easy to understand, is generalized so that changes can be made without revising procedural code, and is smaller than ever. The incremental cost per optimization now is simply (1) a small integer array and (2) a single assignment statement. As a consequence, much more optimizing is now being done.

Small C performs optimizing activities both in the expression analyzer and in the back end. We saw in Chapter 26 that the analyzer lead-in function **test()** optimized tests against the value zero. Also, we saw that constant (sub)expressions were evaluated at compile time, rather than run time, and loaded with a single instruction. Also, subscripting by zero resulted in the total elimination of the subscripting code.

In this chapter, we deal with the *peephole* optimizer in the compiler's back end. Its purpose is to scrutinize the staging buffer as it is dumped to the output file. On finding cases of inefficient code, it makes revisions before passing the code on for translation and output.

Mature optimizing compilers do much more optimizing than Small C. The main difference is that they optimize the whole program, whereas Small C only optimizes expressions. At first this may seem very limiting, but when we consider what a large part of any C program expressions account for, then it becomes apparent that optimizing just expressions can pay off handsomely. Essentially, all that is not covered is the "flow of control" code—the "glue" that holds the expressions together.

Now, before going into the optimizer proper, we look first at its data structures. Having done that, without even looking at its functions, we will fully understand what the optimizer does and how to make it do other things as well.

Optimizer Data Structures

The Staging Buffer Since the peephole optimizer reads and modifies the staging buffer, it is essential for us to have a clear picture of what that buffer contains. Essentially, the staging buffer is just an array of integers; several global pointers address it. **Stage** points to the first word of the buffer. **Snext** is used in two ways. When code is being generated in the buffer, it points to the next available entry; that is, the one beyond the last one made. When the buffer is being optimized, translated, and dumped to the output file, **snext** is reset to the start of the buffer, and thereafter steps forward as each p-code is processed. **Stail** is used to mark the end of data in the buffer while optimizing, translating, and output are being performed. Finally, **slast** marks the physical end of the buffer for overflow detection.

Each entry in the buffer consists of two words—a p-code and a value associated with it. Not every p-code uses its value, but every p-code has one. Each p-code is a small integer value which uniquely identifies an assembly language string by acting as a subscript into an array of string pointers (**code[]**).

The Code[] Array We have already seen the **code**[] array in Chapter 21. Table 21-4 lists the p-code subscripts into **code**[] and the translation strings to which the designated elements point. We covered the translation process in that chapter, but since it pertained to optimizing, we postponed the discussion of the first byte of each string until now.

The first byte of each translation string, which we shall call the p-code's *property* byte, is written as an octal escape sequence. The reason is to make its division into bit fields more obvious. The purpose of this byte is to supply the optimizer with information about the p-codes that enter into its decision-making process. Six questions are answered by each property byte; they are:

1. Does the p-code generate a commutative operation?
2. Does the p-code push something onto the stack?
3. Does the p-code use the primary register?
4. Does the p-code destroy the primary register?
5. Does the p-code use the secondary register?
6. Does the p-code destroy the secondary register?

The encoding of this information is illustrated in Figure 27-1. The named condition is true when the corresponding bit is set to one.

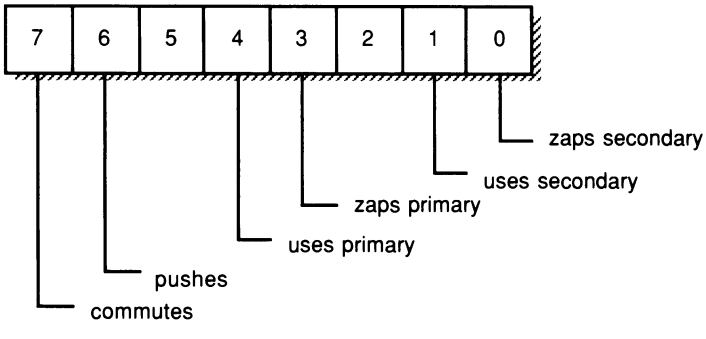


Figure 27-1. P-code Property Byte

To provide access to these bits, six masks, as revealed in Table 27-1, are defined in file **CC4.C**.

Symbol	Value	Selected Bits
PRI	0030	primary register bits
SEC	0003	secondary register bits
USES	0011	use bits for both registers
ZAPS	0022	zap bits for both registers
PUSHES	0100	push bit
COMMUTES	0200	commutation bit

Table 27-1. Masks for Decoding P-code Property Bytes

One of the first two masks is used in combination with (bitwise AND) one of the second two (and the property byte) to determine the effects of a p-code on one or the other of the registers. This arrangement permits **isfree()** to be written independently of the register whose availability it tests. The last two are directly ANDed with the property byte to determine a p-code's stack and commutation properties.

The Seq##[] Arrays **CC4.C** contains declarations for about 50 initialized integer arrays bearing the name **seq##**, where **##** is a sequence number that distinguishes between the arrays. Each array specifies a single optimization case by supplying two kinds of information: (1) a sequence of p-codes to be optimized and (2) a sequence of instructions to the optimizer telling it exactly what to do. In addition, the first word of each array serves as a counter for the number of times the optimization was applied during the compile run.

Recall from Chapter 21 that if **CC4.C** is compiled with the symbol **DISOPT** defined, then, at the end of the run, the compiler displays optimization frequency figures for use in determining whether or not a given optimization is being used enough to make it worth having in the compiler. Naturally, this would only be done in interim versions of the compiler while performing optimizer development. Whether or not the counts are displayed, the optimizer accumulates them in the first word of these arrays.

After the frequency counter, each **seq##[]** array is partitioned into two parts—a *recognition pattern* followed by a set of *optimizing instructions*. Each partition is terminated with a word containing zero.

Each word in a recognition pattern corresponds to an *entry* (two words) in the staging buffer, beginning at the present value of **snext**. For the most part, these patterns contain specific p-codes. For example, one array begins with

```
0,POINT1m,GETw2n,ADD12,MOVE21,0,
```

The leading zero is the frequency counter. Following that is a pattern of four p-codes terminated by zero. When the optimizer sees this pattern of p-codes, at **snext** in the buffer, it looks beyond the terminating zero for instructions.

As we have noted, in matching a pattern to the staging buffer, the optimizer begins at the entry indicated by **snext**. Thereafter, it advances one word in the pattern for two words in the staging buffer. In each case, the current word in the recognition pattern is compared to the first (p-code) word in the current buffer entry. The p-code's value does not enter into the matching process.

Besides actual p-codes, the recognition pattern may also contain what, for lack of a better word, we shall call *metacodes*. These are special codes (that never occur in the buffer) that direct the optimizer to inspect the buffer for specific conditions. These metacodes are defined symbolically in **CC4.C** where they are given large values that cannot conflict with genuine p-codes. These recognition metacodes are described below:

any

This metacode matches any p-code in the staging buffer—a "wildcard."

_pop

This metacode instructs the optimizer to look ahead in the staging buffer for a **POP2** that matches the current stack level. To ensure that the correct pop is found, the optimizer must overlook as many pops as pushes it finds along the way. Failure to find the pop is considered a mismatch.

pfree

This metacode has the optimizer verify that, at the current point in the buffer, the primary register is free. It is considered to be free if its value is not used by any p-code before it gets zapped by a p-code. It is also considered to be free if it is not used before the end of the buffer is reached and the value of the expression

is immaterial to the program—as when the expression stands alone as a statement in its own right. This metacode matches only if the primary register is free.

sfree

This metacode has the optimizer verify that, at the current point in the buffer, the secondary register is *free* (if its value is not used by any p-code before it gets zapped by a p-code). It is also considered to be free if it is not used before the end of the buffer is reached. The context of the expression is not important because the value of the expression is returned in the primary register, not the secondary register. This metacode matches only if the secondary register is free.

comm

This metacode matches only if the current p-code in the staging buffer has the commutative property—i.e., the p-code performs a binary operation and it does not matter which register (primary or secondary) the operands are in.

The second part of each **seq##**[] array, the *optimizing instructions*, tells the optimizer what actions to take in revising the staging buffer. As with the recognition part, this part is terminated by a word containing zero. Also, like the preceding part, the elements of this part of the array consist of either actual p-codes or metacodes. The difference is that, in this case, the optimizer is writing to the staging buffer rather than reading it.

It is important here to understand how **snext** is used during this phase of the optimizing process. We said earlier that, after evaluating an expression, **snext** is reset to the beginning of the staging buffer, from which it advances to each p-code, designating it for translation and output. Before processing each p-code, however, **dumpstage**() enters a loop in which it calls **peep**() once for each **seq##**[] array. **Peep**() attempts to apply the specified **seq##**[] array to the staging buffer at the point designated by **snext**. The important thing here is that when **peep**() receives control, **snext** points to the current entry in the buffer. All preceding entries have been processed (translated and output), and all following entries have yet to be processed. It follows that the manner in which **peep**() manipulates **snext** is crucial.

This brings us to a very important difference between the recognition and

optimization phases of **peep()**. In the first phase **snext** is not changed. In the second phase, however, if the new code sequence is shorter than the original, **snext** is advanced to account for the difference. More specifically, the forward-most part of the original code sequence is revised and **snext** is repositioned to the beginning of the revised sequence, bypassing the unused p-codes at the rear of the original sequence. To summarize this concept, **peep()** adjusts **snext** during its optimization phase, and further processing resumes from that point.

It should be understood that, after applying an optimization, the optimization loop in **dumpstage()** is restarted from the beginning at the current setting of **snext.** After each successful optimization, *all* of the optimizations are attempted once again on the chance that the revised sequence can be further optimized.

Another difference between the recognition and optimization parts of the **seq##[]** arrays is that, in the latter part, each metacode is divided into two bytes—a high-order *code* byte and a low-order *value* byte. We shall define these metacodes and their values momentarily.

P-codes are distinguished from metacodes by the fact that the high-order byte is always zero for a p-code, but never for a metacode. As an instruction, a p-code tells the optimizer to write the p-code itself into the current buffer entry (as designated by **snext**).

On the other hand, a metacode may tell the optimizer to take any of several different actions. The optimization metacodes are defined in **CC4.C** as indicated in Table 27-2.

Symbol	Value	Optimizer Action
go	0x0100	go n entries left or right
gc	0x0200	get code from n entries away
gv	0x0300	get value from n entries away
sum	0x0400	add value from nth entry away
neg	0x0500	negate the current value
ife	0x0600	if value == n do commands to next 0
ifl	0x0700	if value < n do commands to next 0
swv	0x0800	swap value with value n entries away
topop	0x0900	move lcode and current value to POP2

Table 27-2. Optimization Metacodes

Notice that in each case the non-zero part of the code is in the high-order byte of a constant word. This reserves the low-order byte for a value that can be combined with the code by writing something like **go|2**, which yields **0x0102**. As it turns out, we must also be able to combine metacodes with negative values, but writing **go|-2** would overwrite the high-order byte with ones (the extended sign bit). So, to facilitate the writing of negative values, the symbols **m1**, **m2**, **m3**, and **m4** are defined in **CC4.C** as **0x00FF**, **0x00FE**, **0x00FD**, and **0x00FC**, respectively. Now, writing **go|m2** will yield **0x01FE**, which preserves the code byte. It follows that when **peep()** accesses the value byte, it must do it with sign extension; otherwise, it would see a positive number in all cases. Only four negative values are defined, and only four are needed.

Finally, since it seemed offensive to write positive values directly and negative values symbolically, an additional set of four positive symbols (**p1**, **p2**, **p3**, and **p4**) was defined to provide symmetry.

As we look through the **seq##[]** arrays, we shall see numerous examples of optimization metacodes, all written with the | operator, and many with these minus and positive symbolic constants. In the descriptions that follow, the symbol ***n*** refers to these minus or positive values. The optimization metacodes are:

go

This code tells **peep()** to adjust **snext** (the current position in the staging buffer) relative to its current value. So, **go|p3** moves it ahead (the positive direction) by three entries. Since each entry consists of two words, **snext** is actually incremented by six. Also, since it is a *word* pointer, the compiler ensures that it gets incremented by twice that amount. Conversely, **go|m2** adjusts **snext** backward by two entries. The value of **snext** is important in two ways: all of the other metacodes use it as a reference point and, the final value of **snext** (when **peep()** is finished) determines where further optimizing and output processing resumes.

gc

This code *gets the code* from the buffer entry ***n*** entries away and copies it into the current entry. In other words, it fetches the p-code from the designated buffer entry into the current entry. Thus, **gc|m3** copies the p-code from three entries back over the current p-code.

gv

This code *gets the value* from the buffer entry ***n*** entries away and copies it into the current entry. In other words, it fetches the value from the designated buffer entry into the current entry. Thus, **gv|p2** copies the value from two entries ahead over the current value.

sum

This code calculates the *sum* of the current entry's value and the one ***n*** entries away. The result replaces the original value of the current entry. Thus, **sum|m1** tells **peep()** to add the value from one entry back to the value of the current entry.

neg

This code tells **peep()** to arithmetically *negate* the value in the current entry of the staging buffer. Thus, the value in the current entry is replaced by its two's complement.

ife

This code and the next one give **peep()** some decision-making ability, based on the value of the current entry. In this case, if the value *equals* ***n***, then the following metacodes are performed until a zero code is found. At that point, optimizing ceases, and the remainder of the array is ignored. If the value does not equal ***n***, however, then the following metacodes are bypassed until a zero code is found, and optimizing resumes with the following metacode. Thus the sequence,

```
ife|0, go|p1, 0, go|p2, ...
```

would test the value of the current staging buffer entry for zero. If it is zero, then **go|p1** is performed, moving the current position one entry forward, and optimizing ceases. Otherwise, metacodes are skipped until the next zero, after which **go|p2** and whatever follows is performed. If **ife** is used at the end of the **seq##[]** array, then two zero words must terminate the array. For example,

```
ife|p2, go|m1, 0, 0
```

illustrates the proper termination of such an array.

ifl

This code tells **peep()** to test the value of the current entry for *less than* ***n***. If it is, then the following metacodes are performed until a zero code is found. At that point, optimizing ceases, and the remainder of the array is ignored. However, if the value is greater than or equal to ***n***, then the following metacodes are bypassed until a zero code is found. **Peep()** then resumes with the following metacode. Thus the sequence,

```
ifl|m1, go|p1, 0, go|p2, ...
```

would test the value of the current staging buffer entry for -2, -3, and so on. If true, **go|p1** is performed and optimizing ceases. Otherwise, **go|p2** and whatever follows is performed. If **ifl** is used at the end of the **seq##[]** array, then two zero words must terminate the array.

swv

This code tells **peep()** to *swap values* between the current entry and the one ***n*** entries away.

topop

This code should be read *to pop* (not top op). It must be used in conjunction with a **_pop** metacode in the recognition phase. In that case, when **peep()** sees this code, it has already located a **POP2** somewhere ahead in the buffer. The purpose of this instruction is to convert that **POP2** into a different p-code. The idea is to replace something like

```
GETw1m,PUSH1,...,POP2
```

with

```
...,GETw2m
```

To accomplish this, it is necessary not only to change the p-code of the **POP2** to **GETw2m**, but, since the value with the **GETw1m** designates a label number, that same label number must be associated with the new **GETw2m**. Therefore, **topop** assigns an arbitrary p-code (specified by ***n***) to the **POP2** that **_pop** found, and also moves the value of the current entry to that same entry.

```
POINT1s, PUSH1, pfree, _pop, 0,
topop|POINT2s, go|p2, 0
```

has the following effect. First, if the staging buffer (beginning at **snext**) contains **POINT1s** followed by **PUSH1**, if the value in the primary register (as established by **POINT1s**) is not used, and if there is a **POP2** (corresponding to this **PUSH1**) somewhere ahead in the buffer, then the optimizing is performed. In that case, **POINT2s** replaces the original **POP2** (wherever it is) and the value of the current entry (**POINT1s** since **snext** has not yet been adjusted) is also copied to that same entry. Finally, **snext** is moved two entries forward to bypass **POINT1s** and **PUSH1**.

It should be clear that the **seq##[]** arrays actually contain an optimizing "language" that **peep()** interprets. It is this feature that gives **peep()** its generality and allows us to understand its behavior without knowing its logic. Adding other optimizing cases is essentially just a matter of creating additional **seq##[]** arrays.

There is nothing sacred about the particular metacodes in this language. They simply represent facilities that are sufficient to do the kind of things that are currently being done. Should we need other facilities in the future, then the language can be extended by easily defining other metacodes and revising **peep()** to recognize them. But, even with its present capabilities, this language is sufficient for most of the additional optimizing we may wish to do.

At this point, we should be able to interpret the **seq##[]** arrays without even looking at **peep()**. Consider the following case:

```
ADD1n, 0,
if1|m2, 0, if1|0, rDEC1, neg, 0, if1|p3, rINC1, 0, 0
```

ADD1n adds the constant value associated with the p-code to the primary register. The idea here is to replace this three-byte instruction with one or two single-byte instructions. First, if the constant is less than -2, nothing is done. That failing, if it is less than 0, then it must be -1 or -2. In that case, **rDEC1** replaces **ADD1n** and the constant is converted to a positive value. The result is one or two **DEC AX** instructions. That failing, if the constant is less than +3, then it must be 0, 1, or 2. Zero is not possible because the analyzer never generates that

case. Therefore, **ADD1n** is replaced by **rINC1,** which produces one or two **INC AX** instructions.

The Seq[] Array Now we begin to see how the optimizer actually works. First, however, we must be introduced to the array **seq[]**. The purpose of this array is to help **dumpstage()** direct **peep()** to the individual **seq##[]** arrays. **Seq[]** is an integer array with an element for each **seq##[]** array. Each element contains a pointer to an **seq##[]** array. **Seq[1]** points to **seq01[]**, **seq[2]** points to **seq02[]**, and so on. **Seq[]** is defined in **CC4.C** immediately before **setseq()**, which initializes it at the beginning of the run. Just before the definition of **seq[]** the symbolic constant **HIGH_SEQ** is defined. Its purpose is to determine the number of elements in **seq[]** (**HIGH_SEQ + 1**), and to give **dumpstage()** the last subscript to be used in **seq[]** (**HIGH_SEQ**).

Revising the Optimizer

The purpose in grouping these definitions in front of **setseq()** and behind the **seq##[]** arrays is to localize in the source files everything that needs to be changed when optimizations are added or deleted. Adding an optimization involves three steps:

1. Add a new **seq##[]** array, giving it the next available number in the sequence.
2. Increase **HIGH_SEQ** by one to account for the new array.
3. Add an assignment to **setseq()** for the new array.

Removing an optimization involves four steps:

1. Delete the **seq##[]** array for the optimization in question.
2. Renumber the following **seq##[]** arrays to close the gap.
3. Decrease **HIGH_SEQ** by one to account for the deleted array.
4. Delete the last assignment from **setseq()**.

The Optimizer Functions

Dumpstage() When an expression has been entirely parsed, **clearstage()** is called to empty the staging buffer to the output file. **Clearstage()** in turn calls **dumpstage()** to do the work. **Dumpstage()** first copies **snext** to **stail** to mark the end of the data in the staging buffer. It then copies **stage** to **snext** to initially direct processing to the beginning of the buffer. Then it enters a loop that exits only when **snext** is no longer less than **stail**. With each iteration, (1) an attempt is made to optimize the code sequence starting at **snext**, (2) **outcode()** is called to translate and output the p-code at **snext** (which may have been advanced because of optimizing), and (3) **snext** is advanced by two words (one entry).

Step (1) is what we are interested in. This involves first testing the global integer **optimize** to see whether or not optimizing is to be done in this run. If it is, then another loop is entered that steps through **seq[]**, passing its elements to **peep()** one at a time. With each iteration, **peep()** attempts the optimization to which it is pointed, returning *true* or *false* depending on whether or not the attempt succeeded. If the attempt fails, then the loop continues with another attempt. But if it succeeds, then the loop is restarted from the beginning so that all of the optimizations will have a shot at the revised code. Also, if **DISOPT** was defined when **CC4.C** was compiled, the fact that the optimization succeeded is reported on the screen. This is especially handy when entering source code from the keyboard to test the compiler.

It may seem wasteful to restart the optimizer from the beginning with each successful optimization. It would appear that the pointers in **seq[]** could be carefully arranged so that each optimization of an optimized sequence follows its predecessor. That way, only one pass through **seq[]** would ever be needed. In practice, however, it is very difficult to arrange the cases in the proper order, and it is especially time consuming to keep them in order when changes are made. For that reason, I decided to accept the inefficiency of the present arrangement (which testing has verified to be minor). When we consider what a small part of the overall compiling process is spent in **dumpstage()** and that only a relatively small fraction of the p-codes lead to optimizations, then the cost of restarting the optimization loop appears less menacing.

Peep() **Peep()** is the main optimizing function. It accepts a pointer to a **seq##[]** array and attempts the indicated optimization at the point in the staging buffer designated by **snext**. The body of this function comprises two separate loops. The first determines whether or not the current p-code sequence matches the optimization case (the recognition phase). The second performs the optimization if a match occurs (the optimization phase).

A local pointer **next** is used in place of **snext** during the recognition phase so **snext** will not be disturbed. The **seq##[]** pointer **seq**, being an argument, is used to step through the **seq##[]** array. With each iteration of this loop, **next** increments by two to the next staging buffer entry and **seq** increments by one to the next **seq##[]** element. The heart of the loop consists of a **switch** statement in which the current **seq##[]** element is compared to the recognition metacodes. Failure is taken to mean that the element contains a p-code that must match the current p-code in the staging buffer. If it does not match, then *false* is returned to **dumpstage()** which again calls **peep()**, but for a different optimization. If the p-codes do match, however, the pointers are advanced and the loop repeats.

If the metacode is **any**, then the loop simply continues, providing the end of the buffer has not been reached.

If the metacode is **pfree**, then **isfree()** is called to make the determination. If the primary register is found to be free, then the loop continues.

If the metacode is **sfree**, then **isfree()** is called to make the determination. If the secondary register is found to be free, then the loop continues.

If the metacode is **comm**, the p-code property byte in the p-code's translation string is tested for the commutation bit. If it commutes, the loop continues.

Finally, if the metacode is **_pop**, then **getpop()** is called to look ahead for a **POP2** at the present stack level. On finding one, its buffer position is saved for possible use with a **topop** metacode, and the loop continues.

Should any of these conditions fail, however, the function immediately returns *false* to **dumpstage()**. If the end of the recognition part of **seq##[]** is reached, however, then an optimizable pattern has been found, and control drops into the optimizing phase.

At this point, the frequency counter in **seq##[]** is incremented and the optimizing loop is initialized. The first iteration begins with the first optimizing

metacode. Before attempting to perform the current code, a local variable **skip** is checked to see if codes are being skipped because of an unsuccessful **ife** or **ifl**. If so, then the current code is compared to zero to determine if the end of the skipped sequence has been reached. If it has, **skip** is reset. And, regardless, the loop is immediately continued. Control proceeds beyond this point only if skipping is not in effect.

Next, the current code is tested to see if it is a p-code or a metacode. If it is a p-code, then it is simply copied over the p-code in the current buffer entry and the loop repeats.

If it is a metacode, then its low-order byte is extracted with sign extension into a local integer **n**. After that, a **switch** statement identifies the metacode by looking at its upper byte.

If the metacode is **ife**, then the current p-code's value in the staging buffer is compared to **n**. If they differ, then **skip** is set *true*.

If the metacode is **ifl**, then the current p-code's value is compared to **n**. If it is not less than **n**, then **skip** is set *true*.

If the metacode is **go**, then **snext** (the staging buffer's currency pointer) is adjusted by adding two times **n** (a signed quantity) to it.

If the metacode is **gc**, then the current buffer entry's p-code is replaced by the one **n** entries away.

If the metacode is **gv**, then the current buffer entry's value is replaced by the one **n** entries away.

If the metacode is **sum**, then the current buffer entry's value is replaced by the sum of itself and the one **n** entries away.

If the metacode is **neg**, then the current buffer entry's value is replaced by its two's complement.

If the metacode is **topop**, then the buffer entry located previously by **getpop()** receives **n** for its p-code and the current entry's value for its value.

Finally, if the metacode is **swv**, then the values of the current entry and the one **n** entries away are swapped.

In each of these cases, a local variable **reply** is set *true* to indicate that optimizing took place, and the loop is continued. When the **seq##[]** array has been exhausted, **reply** is returned to **dumpstage()**.

Isfree() **Isfree()** is a small function that decides whether or not one of the registers is free. It receives the mask bits for the register to be tested (**PRI** or **SEC** in Table 27-1) and a p-code pointer (**pp**). In a loop, it steps through the buffer until a determination is made or the end of the buffer is reached. With each iteration the next p-code's property byte is tested. If it indicates that the p-code uses the register, *false* is returned—the register is not free for other uses. If the property byte indicates that the p-code zaps the register, however, the original value in the register was not used and so the register is free for other uses. In that case *true* is returned. If the end of the buffer is reached, then a test is made to determine if the expression being optimized is a stand-alone statement. If not, **usexpr** (a global variable) is *true* (as set by **doexpr()**); the expression's value is needed, so if the primary register is being tested, *false* is returned. If the secondary register is being tested, however, *true* is returned. On the other hand, if **usexpr** is *false*, then the value of the expression is definitely not needed, so *true* is returned regardless of which register is being tested.

Getpop() **Getpop()** looks ahead in the staging buffer for a **POP2** at the current stack level. It receives a p-code pointer **next,** which it increments by two with each iteration of a loop. A local integer **level** is initialized to zero, indicating the current stack level. Thereafter, each push increases it by one and each pop (at any level but zero) decreases it by one.

With each iteration of the loop, three checks are made. If the end of the buffer has been reached, *false* is returned—no **POP2** was found.

If a **POP2** is found, and if **level** is zero, the desired pop has been found so *true* is returned. But if **level** is not zero, it is decremented and the loop continues.

If an **ADDSP** is found, the stack is being adjusted to deallocate function arguments that were pushed earlier. In that case, **level** in decreased by half of **ADDSP**'s value—once for each word being deallocated. Since we are in an expression, this is the only condition under which an **ADDSP** can be generated.

Finally, if the "push" property bit for the p-code is set, then the p-code involves a push instruction, and so **level** is increased.

CHAPTER 28

Further Development

Programs are seldom perfect, and the Small C compiler is no exception. Many of its algorithms can be improved. Much of its code could be organized better and made more efficient. No doubt there are bugs lurking in the darkness. And, of course, many features of the C language have yet to be implemented. So, there is no shortage of improvements that can be made. You may even want to revise it simply for the experience.

Whatever the reasons, you may find yourself using the compiler to create new versions of itself. On the chance that you will, I have included this chapter of suggestions and helpful hints. Chapter 17 explained the procedure for compiling the compiler. Here we look at some of the things you might want to do to it and how you might do them.

Before proceeding, I must confess that the suggestions which follow are untried. Had I tested them, they would no doubt be in the compiler already. In fact, I did install about a quarter of the original recommendations simply because after investigating them, going ahead with the implementation was irresistible.

Furthermore, as every programmer knows, recommending changes is a bit risky. Plans that look good at first have a way of turning sour with implementation. It is hard to think of everything at the beginning. Some details come to light only when the changes are being made or when testing reveals them. I must admit to being especially prone to that kind of error.

So as you work through these projects, understand that you are breaking new ground. Along the way you will learn things about the compiler that I may not have prepared you for. And you will certainly be challenged to find new solutions—hopefully elegant ones. But that is what learning by experience is all about. No amount of talk can replace the lessons learned by actually doing something.

With that disclaimer, I offer the following suggestions for further development.

Test Programs

One project that should precede the others is the development of a suite of test programs for the Small C compiler. A compiler is a complex program. It is easy to mess up one part of it while working on another. So it is important to fully test each new generation of the compiler to ensure that nothing but what was supposed to change did in fact change.

The challenge here is to come up with a set of programs that fully exercises the compiler's capabilities. Every sort of normal and exceptional condition should be tested. Doing this properly requires a thorough knowledge of the compiler and a large dose of ingenuity.

The test programs should be written in such a way that it is obvious when they fail. In addition to reporting on their progress, they could also be written to keep track of the results for a summary display at the end of the run. Since a large number of test cases will be tried, this would help the programmer determine quickly whether or not problems occurred.

The test programs should be augmented with additional tests whenever compiler bugs that were not previously detected are discovered.

Eliminate Quote[]

This exercise does little more than acquaint you with the process of making revisions. You might find it a good way to begin.

There is a global character array in CC1.C called **quote[]** which is made to look like a string containing just a quotation mark. Originally, Small C did not support the backslash escape sequences, so this device was used instead.

Eliminate this array and replace the reference to it in **string()** with **"\""**.

Continued Character Strings

One of Small C's shortcomings is that it provides no means of continuing character strings that are too long to fit in one line. The maximum line size is 127 characters, as determined by **LINEMAX** and **LINESIZE** in **CC.H**. If you

have a text editor that supports lines that long, you could easily exceed the width of your screen. In fact, you could recompile the compiler with larger values for these symbols and thereby handle any length line your editor will allow. But that is a poor solution since listings of your programs (depending on the printer) will either truncate or wrap around to column one (throwing off the internal line count of whatever print utility you may be using). This solution is especially bad because it denies you control of your program's appearance.

The standard C solution to this problem is to place a single backslash (\) at the physical end of the source line (without trailing white space). In that case, the backslash and the newline are ignored and the string continues with the first character of the next line. Thus,

```
"This is the first part a\
nd this is the last part."
```

(where the leading quote is in column one) is the same as

```
"This is the first part and this is the last part."
```

Obviously, this solution is crude and gives the programmer very little control over the appearance of his program. To overcome that limitation, some implementors have deviated by having the compiler also discard leading white space on the continuation line. In that case, the line is continued with the first graphic character on the continuation line.

Current practice, however, is to have compilers concatenate into a single string any series of strings that are separated by nothing but white space (newlines included). Thus,

```
"Part one, "
"part two, " "part three."
```

would be seen by the compiler as

```
"Part one, part two, part three."
```

Clearly, this approach is much better than the previous two. It gives the programmer complete control, and it makes the programmer's intentions explicit. There is no guessing as to whether or not the compiler is overlooking leading white space in a continuation line.

Finally, and best of all for Small C, this solution is the easiest to implement. You may recall that the Small C preprocessor operates on a line by line basis. When it finds a string's leading quote, it searches for the trailing quote in the same line and complains if it does not find it. Making the front end of the compiler able to deal with a single string that spans source lines would be difficult.

Therefore, since it is easier and is in keeping with modern practice, I recommend the string concatenation approach. To implement it, only one small function **string()** should require alteration. Presently, it contains a single loop in which it obtains the characters of a string and stores them in the literal pool. On exit from the loop, it caps the string with a null terminator. Prior to entering the loop, it passes the initial value of the literal pool offset **litptr** back to the caller as a side effect. This will be used in locating the string in the pool when it is dumped at the end of the current function.

String concatenation can be easily added by enclosing the current loop in an outer loop. This loop might best be written as a **do...while** statement. The **...** includes the existing loop, as well as the **gch()** that bypasses the trailing quote. The **while** condition could be the expression:

```
(match(quote))
```

On completing the first string and bypassing its trailing quote, any white space (including newlines) would be skipped and the next token would be tested. If it turns out to be another quote, then it is bypassed by **match()** and the expression yields *true*, which restarts the inner loop. Be sure that the passing of **litptr** occurs before the outer loop, and the terminating of the string with a zero byte occurs after it.

Testing this change should be extremely simple. Just execute the compiler without parameters so that it inputs from the keyboard and outputs to the screen. Then initialize some character arrays and pointers with simple and concatenated

strings. After that, start a function in which you reference character strings. If the generated code is right, you are finished. Remember to try boundary conditions like empty strings, no white space between concatenated strings, and so on.

Better Code for Integer Address Differences

When the compiler subtracts one address from another, the result must be interpreted as the number of objects (not necessarily bytes) lying between the addresses. Subtracting one character pointer from another automatically yields the desired result; but subtracting one integer address from another would yield twice the desired value if it were not scaled down by a factor of two. As an example, consider

```
int *ip1, *ip2;
main() {
  ip2-ip1;
  }
```

The code produced by the expression is:

```
                    MOV BX,_IP1
                    MOV AX,_IP2
                    SUB   AX,BX

SWAP12 0   ->       XCHG  AX,BX
GETw1n 1   ->       MOV   AX,1
ASR12  0   ->       MOV   CX,AX
                    MOV   AX,BX
                    SAR   AX,CL
```

where the p-codes that perform the division by two are shown on the left, together with their respective values. Five assembly language instructions are used to shift the answer right by one bit to accomplish the division. This is clearly wasteful. How much better it would be to generate:

```
                    MOV BX,_IP1
                    MOV AX,_IP2
                    SUB AX,BX

ASR12  1   ->    MOV CL,1
                    SAR AX,CL
```

instead. This improvement can be achieved easily by modifying **down2()** and the translation string for **ASR12**.

Down2() now generates:

```
SWAP12    0
GETw1n    1
ASR12     0
```

The first two p-codes can be dropped if the translation string for **ASR12** is changed appropriately. Currently, **ASR12** translates through

```
\011MOV CX,AX\nMOV AX,BX\nSAR AX,CL\n
```

which makes no use of the p-code's value. By changing this to:

```
\011?MOV CL,<n>?MOV CX,AX?\n??MOV AX,BX\n?SAR AX,CL\n
```

the translation will take into account whether or not the p-code's value is zero. If it is, then the shift count is assumed to be in AX as usual. A value of one will be taken as a constant shift count, however, and only

```
MOV CL,1
SAR AX,CL
```

will be produced. Besides dropping the first two p-codes generated in **down2()**, the value that goes with **ASR12** must be changed to one.

Be advised that this revision of the ASR12 p-code usage and translation string conflicts with the advice given in Long Integers below. If you intend to

carry out that project, then you should simply define a new p-code for use here. Test this with code like that above. This relatively easy project can be used as a warm-up for the ones that follow.

Better Code for Logical AND and OR Operators

The code generated by the logical AND and logical OR operators can be improved somewhat. The expression **i && j,** where **i** and **j** are globals, generates:

```
    MOV AX,_I
    OR AX,AX
    JNE $+5
    JMP _2
    MOV AX,_J
    OR AX,AX
    JNE $+5
    JMP _2
    MOV AX,1
    JMP _3
_2:
    XOR AX,AX   <-
_3:
```

and the expression **i || j** generates:

```
    MOV AX,_I
    OR AX,AX
    JE $+5
    JMP _4
    MOV AX,_J
    OR AX,AX
    JE $+5
    JMP _4
    XOR AX,AX   <-
    JMP _5
_4:
    MOV AX,1
_5:
```

Notice in each case that an unnecessary **XOR AX,AX** instruction is generated. When control reaches these instructions, AX is already known to be zero. These can be eliminated by revising s**kim**() appropriately. By testing the argument **tcode** for **EQ10f** or **NE10f**, it can be determined which of these instructions (**GETw1n**, where **n** == 0) should be eliminated.

In the first case, the resulting code

```
    ...
    MOV AX,1
    JMP _3      <-
_2:
_3:             <-
```

can be improved further by eliminating the jump to label 3 as well as the label itself. Likewise, in the second case

```
    ...
    JE $+5
    JMP _4     <-
    JMP _5     <-
_4:
    MOV AX,1
_5:            <-
```

can be improved further by eliminating the two adjacent jumps to labels 4 and 5 as well as label 5. In this case, since **MOV AX,1** occupies 3 bytes, just like **JMP _4**, **JE $+5** (2 bytes) will transfer control directly to label 5, but without referring to the label. Eliminating the first of these jumps is complicated by the fact that **dropout**(), which is called from other contexts too, must be modified. Eliminating the unused labels is not a priority item since they have no effect on the final **EXE** file. They simply take up a little space in the **ASM** file.

Test this from the keyboard first, then create an exhaustive test program. Remember to test all cases that call **dropout**(), not just the logical AND and OR. This is probably a semester-long project for someone who is new to Small C.

Consolidated Type Recognition

There are three places in the compiler where data declarations are parsed. Global declarations are parsed by **dodeclare()**, formal arguments are parsed by **dofunction()**, and local declarations are parsed by **statement()**. Each of these functions has essentially identical logic for recognizing the lead-in syntax to a declaration—the data type specification. They must each recognize:

```
char
int
unsigned
unsigned char
unsigned int
```

to determine one of four data types—**CHR**, **INT**, **UCHR**, or **UINT**. This redundance is not good, but its cost is relatively slight now since so few data types are supported. If other data types are added to the compiler, however, then the cost in code size would become less acceptable. And, of course, there is the inconvenience of having to apply essentially the same revision to three different places.

To improve this situation, I suggest a project in which you consolidate this recognition logic into a single function (say **dotype()**). Have it return one of the four data types on success and zero on failure. Then call the new function from the three places in question. Notice that **dodeclare()** has to handle the special case of **extern** declarations. Is that really a problem?

Testing is simply a matter of throwing all possible syntax variations at the compiler, and verifying that it defines or declares items properly, and making sure that references to them see them properly—as integers or characters, signed or unsigned. This latter test will require devising expressions with operators that have different signed and unsigned versions.

This as basically a warm-up project. If you have learned how to recompile the compiler, this should take only a day or two.

Eliminating the While Queue

You may have wondered, as you studied **dowhile()**, **dofor()**, **dodo()**, and **doswitch()**, why they each maintain a local copy of their current entry in the while queue. The answer is that it makes referencing the fields in the entry more efficient. **Addwhile()** might have returned a pointer to the new entry in the global queue. These four functions could then save it in a local pointer and make their references by subscripting that pointer. But in that case, there would be a local reference (relative to BP) for the pointer and then a subscripted reference from the pointer to the item in the queue. So while it looks wasteful, it is not really all that bad. It runs faster and saves space in the code segment, but takes three more words of space on the stack.

A more significant problem with the present arrangement is the fact that the global while queue restricts the nesting of loops and switches to 10 levels. Why not eleminate the while queue altogether and that limitation with it? Each function that creates a new entry in the queue (the four listed above) already has its own new entry allocated locally on the stack. So why not let the stack frames that nest with these function serve the purpose of the global queue?

The purpose of having a global queue is so that the **break** and **continue** statements can find the proper stack level, exit label, and loop label for the level of nesting at which they occur. **Dobreak()** calls **readwhile()** for a pointer to the lowest entry in the queue, and **docont()** calls it for the lowest level with a loop label (excludes **switch** entries).

This purpose can be served by having **addwhile()** and **readwhile()** work with the local **wq[]** arrays. **Addwhile()** does this already in addition to establishing an new entry in the global queue and loading it. It would simply need to have some code removed so that it only loads the local array. But **readwhile()** must be able to scan the queue backward. This can be accomplished by imbedding a backward pointer chain in the individual **wq[]** arrays of the four **do...()** functions.

First, in **CC.H**, delete the definitions for **WQTABSZ**, **WQSIZ**, and **WQMAX**; define **WQLINK** with an offset value of zero; and increase the offset values of **WQSP**, **WQLOOP**, and **WQEXIT** by one each.

Then, in **CC1.C**, delete the global declarations for ***wq** and ***wqptr**. Add a global declaration for a *head of chain* integer pointer. This will be zero when

there are no active loops or switches, and will point to the last **wq[]** when there are.

Delete the statement in **main()** that allocates space for the while queue.

Modify **addwhile()** to load **wq[WQLINK]** (the current local array) with the global pointer, and then move the address of the current **wq[]** to the global pointer. This will establish the chain that ends with a null link.

Next modify **readwhile()** to follow the chain rather than stepping backward in the global array. This would mean passing it the new *head of chain* pointer instead of **wqptr** in **dobreak()**. Modify **docont()** in a similar way.

Delete **delwhile()** and modify the four **do...()** functions to replace their call to **delwhile()** with an assignment of their own **wq[WQLINK]** to the global pointer. This will shorten the chain.

The end result will be a slightly smaller compiler with one less restriction.

Be sure to think your way through this revision until you are convinced that it will work in every situation. Test this revision first by throwing various nested loops and switches at the compiler and verifying that they all continue and exit properly. Also verify that the stack gets adjusted properly. You will need to declare local variables within the nested blocks to check that out. Next compile a program, like AR.C on the distribution diskette, and verify that it runs. If there are problems in this patch, they will most certainly show up by the program going berserk. Finally, recompile the compiler as the acid test.

Nesting Include Files

As you may recall from the description of **inline()** in Chapter 22, only a single include file can be active at one time. That is because there is only one variable for holding the file descriptor of include files—**input2**. **Doinclude()** processes **#include ...** directives by simply opening the named file, assigning its file descriptor to **input2**, and killing the line containing the include directive. Thereafter, **inline()**, seeing that **input2** is not equal to **EOF**, reads from it instead of **input** (the primary file's descriptor). When **input2** expires, **inline()** resets it to **EOF** and reverts back to using **input** again.

So what happens when an included file contains an **#include ...** directive? That's right, **input2** gets overlaid with the new include file's descriptor and the

original include file is completely forgotten. In effect, two include files have been chained, just as MS-DOS batch files can be chained. In this case, however, it just happens to work that way.

Now, how can we make Small C honor **#include ...** directives properly? How can we make it resume with the first include file when the one it includes is finished? The two obvious approaches are (1) to replace **input2** with an array of file descriptors, and (2) use recursion. Recursion is desirable because it would impose no limitation on the levels of nesting. As it turns out, however, Small C's front end is not designed for easy implementation of that idea. Therefore, I suggest the array approach.

You will need to declare a global array with as many levels as the number of active include files you wish to allow—say 10. Also declare a global subscript (say **inest**) into the array that designates the level of include nesting; you might initialize it to -1 so that it can be used directly as a subscript once it has been incremented. Anytime it is greater than or equal to zero, one or more include files have temporarily superseded the primary input file, and **inest** locates the current file descriptor.

Obviously, you will need to delete **input2**. **Doinclude**() will need revision to (1) detect nesting overflow, (2) increment **inest** to the next nesting level, and (3) put the new file descriptor in the array. **Inline**() will have to be changed to (1) test **inest** instead of **input2** to determine whether or not an include file is active, (2) subscript into the array for the include file descriptor, and (3) decrement **inest** when the end of an include file is reached.

These changes will endow the compiler with the ability to nest include files. By now, however, it has probably occurred to you that the logic in **inline**() can be streamlined further by not treating the primary file and the include files differently. Why not go all the way and eliminate **input**, too? Let the first element of the descriptor array refer to the primary file, and the higher-lying elements the include files. In that case, **inest** would still be initialized to -1 to tell **inline**() that the primary file has not yet been opened. When it is greater than zero, an include file is to be used. Finally, another global variable **eof** can also be eliminated by letting a negative value in **inest** indicate the end of input text. You will need to use your text editor to search for all references to **eof** in the compiler and fix

them up appropriately.

Study **inline()** and other front end functions to decide whether -1 in **inest** can serve both to indicate that the primary file is not open initially and to indicate the end of file condition. If not, then perhaps -2 can serve for the end of file condition.

Testing these changes should also be easy. You might try keyboard input initially, then verify with files that everything works properly. Again, try boundary conditions like null include files, a primary file with nothing but an **#include**, and so on.

Void Argument Lists

A trend in the current movement toward a standardized C language is to use the keyword **void** in places where, by design, no value is to be found. This compiler already accepts **void** in front of function definitions to document the fact that the function is not supposed to return a value. But another use of the keyword is in the parentheses that normally contain the list of formal arguments. When that list is intentionally empty, the word **void** can be written to make it clear that the omission is by design.

The function to be modified for this feature is **dofunction()**. You will find there a loop controlled by:

```
while(match(")") == 0)
```

where the formal argument list is processed. All that is required is to make the loop conditional on not finding **void**. If **void** is found, then you must enforce the closing parenthesis. Consider using **match()** and **need()** in your new code. This project should be easy to test with keyboard input to the compiler.

Prototype Function Declarations

Another trend in modern C compilers is to have the compiler verify that the actual arguments passed to functions are of the correct type and number. Failure to properly pass arguments is probably the most frequent error committed by C programmers. So this enhancement goes a long way in helping programmers avoid wasted time.

In order to do this, of course, the compiler must know, at the point of a function's call, what arguments it requires. And, since C allows forward references, that information is not always available. In such cases, the compiler could not police the programmer's arguments.

To overcome this limitation, modern compilers support a feature called *function prototyping*. The idea is that a function may be *declared* early in the source file even though it may be *defined* later. The declaration would specify, in place of formal argument names, the types of the arguments. The result is a *prototype* (or *template*) of the way the function is to be called. Of course, the same device could also be used with external functions.

So, at two points the necessary information should be collected. If a prototype declaration comes first, then it specifies the number and type of the arguments. But if there is no prototype declaration, then the function's definition provides the information. If both are present, then the function's definition should be policed according to the prototype declaration. Finally, function calls are policed only if the information has been collected—whether from the prototype or the definition.

The first problem is to decide how to store the argument information. Obviously, it must either be in or tied to global symbol table entries. The problem is that, since the number of arguments is variable, a variable length list is needed. Therefore, it would be best to have a pointer in the symbol table designate the head of a list of data descriptors that are stored in a buffer somewhere. The descriptors can be small non-zero numeric values, and a zero value could terminate the list. Byte strings would suffice. As it turns out, there are two unused fields in symbol table entries for functions—**SIZE** and **OFFSET**. The latter is already named appropriately for use as an offset into a buffer of argument types.

To implement the buffer, you will need to declare a global character pointer (say **argtype**) that points to the buffer, for which you must allocate memory in **main()**. In addition, you will need an integer (say **argnext**) to keep track of the next available position in the buffer. And, you will need a constant (say **ARGTSZ**) to specify the size of the buffer and designate its end. In addition, you will need a series of constants for the values that will be stored in the buffer.

At this point you must decide how strictly you will police the programmer. If you are too strict, he will not want to use the compiler. If you are too easy, you will miss true errors. For instance, if an argument is declared to be a character variable, should an integer be acceptable? After all, characters are promoted to integers when referenced, and both appear as integers when they are passed on the stack. Why shouldn't the programmer be allowed to pass integers to a function that is written for characters and vice versa? On the other hand, isn't it likely that such a mismatch is an oversight?

One way to handle such decisions is to throw them back on the programmer, as many compiler writers do. Provide a command-line switch that lets the programmer override the checking, or control its severity. Personally, I never use those options. They are too much trouble to remember; I prefer to have one way of invoking the compiler which I routinely use. Therefore, I recommend making the checks fairly lenient, and flagging only those errors that would most likely cause a program to blow up. After all, even with exact matching, you will not catch all of the programmer's errors. He can still pass the wrong integer as an integer argument, for instance. I suggest that you debate this issue and justify your decision.

In any case, I recommend defining the following symbolic constants

```
ARG_C       1     /* character */
ARG_CU      1     /* unsigned character */
ARG_I       1     /* integer */
ARG_IU      1     /* unsigned integer */
ARG_CA      2     /* character array */
ARG_CAU     2     /* unsigned character array */
ARG_CP      2     /* character pointer */
ARG_CPU     2     /* unsigned character pointer */
ARG_IA      3     /* integer array */
ARG_IAU     3     /* unsigned integer array */
ARG_IP      3     /* integer pointer */
ARG_IPU     3     /* unsigned integer pointer */
ARG_F       4     /* function address */
```

where those symbols with the same values are accepted as a match. You can make the rules stricter simply by renumbering the symbols. Notice that arrays and pointers of the same type match; that is because arrays are passed as addresses on the stack and so are effectively pointers as far as the called function is concerned.

Now comes the hard part—revising the logic. To support prototype declarations, you must work on **declglb()**. At the point where it recognizes a function, it sets **id** to **FUNCTION** and calls **need()** to enforce the closing parenthesis. Between these, you must insert a loop that (1) recognizes a list of declarators without names—for example, (**char, int *, unsigned, char []**)—(2) deciphers them, and (3) stores in **argtype** one of the symbolic constants. In each case **argnext** must be incremented. And, at the end, a null value must be stored. Also, the initial value of **argnext** must be preserved so it can be fed to **addsym()** as the offset value (currently zero for functions). The loop must also verify that **argtype** does not overflow. Consider, in this loop, calling a function (say **declarg()**) to perform the recognizing and storing operations for each argument. Not only will it keep **declglb()** cleaner, but you will have use for it shortly.

Now, you must perform a similar task in **dofunction()**. And here a decision must be made. Modern C compilers support a simpler way of declaring formal arguments than the method now supported by Small C. While still supporting the original syntax, they also permit functions to be written like

```
func(char ch, int *ip, unsigned u, char ca[]) {
...
}
```

where each formal argument is written as a declaration. Now, we have just written a function **declarg()** that will recognize everything about these arguments except their names. It could be upgraded to either look for a name or not, depending on an argument it receives. It could then be called from **declglb()** or **dofunction()**. Basically, this would involve adding to it the functionality found in the argument loop of **dofunction()** and in **doargs()**. To determine whether the old or new style syntax is being used, this function could return a success or fail-

ure indication. If it fails on the first argument, the old style syntax could be assumed. On success, the rest of the arguments could be parsed using the new function. I leave the details to you.

With this arrangement, the programmer would call for argument verification either by writing a prototype declaration or by using the modern syntax. In addition, however, the existing function **doargs()** could be extended to store information into **argtype**. That would cover all of the bases.

At the other extreme, the validation information could be collected only in prototype declarations. That would save a lot of revising of the compiler. It would also give the programmer a lot of control, but it would require every validated function to have a prototype declaration. Since most C programs are written in a "forward reference" style, prototype declarations would be needed anyway, so not much additional work would be demanded of the programmer. You might first try gathering the validation data only in **declglb()**. Then, after you have argument validation working, decide whether also having **dofunction()** gather validation information is worth the trouble. Or you could do the new style of formal arguments as a separate project, and at that time gather validation information.

At this point you are ready to do the actual validating. There are two candidates for this operation. First, if a prototype declaration created a symbol table entry and collected validation information in **argtype**, then when the function is defined by **dofunction()** the compiler should complain about differences. And, of course, function calls that are handled by **callfunc()** should definitely verify arguments.

The verification process should simply be a matter of stepping down the function's string in **argtype** as each argument is parsed. In each case, if the identity and data type do not match the stored code, then **error()** is called to complain. If the end of the string is reached before the arguments run out, then too many arguments are being passed. Conversely, if unreferenced codes remain in the string after the arguments end, then too few arguments are being passed.

If no validation data exists for a function, then no validating should be attempted. This can be detected by making the offset in the symbol table entry zero in that case. But be sure to skip the first byte in **argtype** so as to guarantee

that the first function will be checked. Another approach would be to let the offset be an actual address instead of an offset into the buffer. Since all addresses are guaranteed by the compiler to be non-zero, that should work fine. In addition, it would reduce subscripted references to the buffer to simple pointer references.

Finally, there is the problem of how to gracefully handle functions that take a variable number of arguments. The simplest solution is to gather validation data only in protocol declarations, then avoid writing such declarations for those functions—that is, never validate them at all. Microsoft handles this by allowing the trailing argument in a prototype declaration to be written as .... This tells the compiler that an unspecified number of additional arguments may follow. Of course, they could not be verified.

The error messages generated by this mechanism should be treated as warnings only; that is, the error message should be generated, but the compiler should function just as though no verifying was being done. The generated code should not be affected. If **argtype** overflows, then after complaining once, the compiler should simply verify those functions for which data was gathered and quit gathering data for the functions that remain.

As you can see, this is a sizable revision. And, for obvious reasons, I have left a lot of the design decisions and details to you.

In my opinion, this project is a bit much for a semester. It might be broken into two parts—support for prototype declarations with argument verification of function calls and definitions, and support for the new formal argument syntax.

Backward Jumps with Block Locals

The restriction that local declarations in nested blocks cannot coexist in the same function with **goto** statements can be relaxed somewhat. Backward **goto** references can be allowed. This can be done by having **addlabel()** save **csp** in the symbol table in either the **TYPE**, **SIZE**, or **CLASS** field, all of which are unused by labels. To distinguish between backward and forward references, another unused field could be used as a flag. Set the flag one way when the label is added to the table by **dogoto()**, the other way when the label itself is parsed by **dolabel()**, regardless of whether or not the entry already exists. Before generating **JMPm**, have **dogoto()** generate **ADDSP** (passing it the saved value of **csp**).

If **csp** is saved in the symbol table in both cases, then a forward reference will produce an adjustment of zero (see **gen()**) which will simply be ignored when **ADDSP** is translated to ASCII by **outcode()**. Therefore, there is no need to avoid generating **ADDSP** for forward references.

If the reference is forward and if **nogo** is *true* (set by **statement()** because block locals were declared), issue an error message. In **dogoto()** the original error message must be eliminated, and **nogo** must be set only if the reference is forward. In **declloc()** the error message should be changed to better describe the violation.

Before installing this revision, think it through. Be sure you understand why **ADDSP** has to be generated, why it works with backward references, and why forward references cannot be allowed. And, convince yourself that the changes you plan will cover all the bases. This is probably a half-semester project.

Word Alignment

Machines with 16- or 32-bit data buses, are able to fetch or store word-length objects with a single memory access provided the objects are aligned on word boundaries. However, words with odd addresses always require two accesses in machines with 16-bit buses, and require two accesses in about half the cases in machines with 32-bit buses. Obviously, a performance bonus can be obtained in these machines by ensuring that 16-bit objects fall on word boundaries in memory. Currently, the Small C compiler makes no effort do this. Aside from declaring global pointers, integers, and integer arrays first, the programmer can do nothing about function arguments and locals, he.has no means of ensuring word alignment.

Word alignment can be achieved easily by inserting an extra byte before each 16-bit object that would otherwise fall on a byte boundary. But there are three cases to consider—global objects, arguments, and local objects.

In the first case, a global integer (say **glc** for *global location counter*) can be defined which increments by the size of every global object that gets defined. Then when an integer, an integer array, or a pointer (to integer or character) is about to be defined, **glc** is first checked. If it is odd (low-bit set), a byte is defined by calling

```
gen(BYTEn, 0);
```

first. Put this fix in **declglb()** before it calls **initials()**. Also, don't forget to bump **glc** for the filler byte.

Arguments are passed on the stack. Since all arguments are 16 bits long, the only thing to do is ensure that the stack pointer is on a word boundary before starting to evaluate the actual arguments in **callfunc()**. In this case, we already have a location counter, the *compiler-relative stack pointer* (**csp**). If it is found to be odd on entering **callfunc()**, then call

```
gen(ADDSP, csp - 1);
```

Gen() takes care of adjusting **csp** for you.

Now, notice that the last statement in **callfunc()** restores the stack to its original value which it calculates as **csp + nargs**. That fails to take the alignment byte into consideration, so perhaps we should save **csp** on entry to **callfunc()**, then restore it from the saved value instead.

The last case deals with local declarations of integers, integer arrays, and pointers. The pertinent function is **declloc()**. This change is easy since the stage is already set. There is already a counter called **declared** that determines the stack frame offset for each local and also accumulates the total number of bytes allocated. When **statement()** finds its first executable statement in a block, it generates **ADDSP** to adjust SP for the number of bytes indicated by **declared**. So all that needs to be done is to find the place in **declloc()** after the object's identity and data type have been determined and before **declared** is adjusted to reflect the offset for the current object, and make an alignment adjustment if necessary. Just before

```
declared += sz;
```

test **id** and **type** to make the decision. Then if alignment is desired and **declared** is odd, bump it by one. Notice that aligning locals this way only works if arguments are also aligned on the stack.

Since alignment is somewhat wasteful of space, you could define a *no alignment* command-line switch **-NA** to disable this feature at the programmer's discretion. Alternatively (or in addition), you could look for a defined symbol in the program (say **NOALIGN**) to override alignment. This has the advantage of allowing the programmer to specify, once in a program, that there is to be no alignment. Different choices can be made for different programs without the programmer having to remember at compile time.

As always, before making the source changes, convince yourself that the revision will work and that all the bases are covered. You should know, for instance, why the stack is on a word boundary on entry to a function and why **declared** contains zero at the same point. But do not stop there. What about entry to inner blocks? Is **declared** zero? Is the stack on a word boundary? If not, how can you guarantee that it will be? (Hint: Look at the point where **statement()** uses **declared**.)

Testing this revision is a bit more difficult than most of the others. If your changes do not work, test programs will most likely run anyway—just slower than they should. First, use keyboard entry to catch the gross errors. Then write a small test program that exercises all three cases (globals, arguments, and locals), with and without alignment adjustments. Compile the program, inspect its **ASM** file, and run it to verify that it has not been corrupted. Finally, obtain a linker map of the program so you know where the globals and functions are. Verify that the globals are properly aligned. Then execute the program with **DEBUG**, or some other debugger, and verify that references to locals generate even-numbered addresses. Be sure to write code in your program that will definitely force some word-length objects (in each of the three cases) to fall on an odd boundary without alignment; after all, you could walk away satisfied having verified nothing but happy coincidences.

One final consideration should not be overlooked. The memory allocation routine **_alloc()** in **CSYSLIB.C** should be modified to ensure that each dynamically allocated block of memory begins on a double word boundary. That way, the user has the option of improving efficiency by organizing heap memory structures in the order: double-words, words, bytes.

This would probably be a nice semester project.

Restarting the Optimizer Loop

In Chapter 27 it was pointed out that each time **peep()** applies an optimization, the list of potential optimizations is rescanned from the beginning, in an effort to improve the optimized code. It was pointed out that it would be more efficient to arrange the optimization cases so that optimized code is fully covered by optimizations that follow in the sequence, allowing the optimizing loop in **dumpstage()** to simply continue on with a single pass through **seq[]**. The justification for restarting the loop was given as ease of maintaining the compiler's optimizer. In this project, you should remake that decision for yourself. Find out how much performance is improved by not restarting the loop and balance that against the difficulty of keeping the optimization cases in sequence so that optimizations are not lost on optimized code.

First, compile a version of the compiler with **DISOPT** defined in **CC4.C**. Then devise a test program that forces every possible optimization case. This will be used to verify that optimizations are not being missed. Also acquire some typical nontrivial programs for use in testing performance under realistic circumstances. **AR.C** is a good choice, or you could even use the compiler itself.

Compile the first program and note which optimizations were applied. Revise the program until you have forced every one to occur. This will be a learning experience. Are there any cases that cannot be forced?

Now, time the compiles of the conventional programs. And make a record of the compiler's speed. Next, reorder the optimization cases. This can be done either by renumbering the **seq##[]** arrays or by changing the way **seq[]** is initialized in **setseq()**. Which is easier? Then, revise **dumpstage()** so that its inner loop is not restarted after successful optimizations. All of this can be done by working with **CC4.C** alone. If you have already recompiled the compiler and saved the four **OBJ** files, then only **CC4.C** needs to be recompiled and assembled.

Now compile the special test program to verify that all of the optimizations are still taking. If not, revise **CC4.C** again. Is it always possible to find a sequence that covers every case? Could a cyclical situation develop?

Now, time the revised compiler on the conventional test programs. Is it faster? How much faster? Is it worth the trouble? This is probably a semester-long project.

Stack Probes

One of the risks associated with CPUs that have stacks is that the stack may overrun its allotted space. Since the amount of space needed by the stack depends on the program's algorithm and the data upon which it operates, it is usually not possible to predict with accuracy exactly how much stack space is needed. Recursive algorithms are especially susceptible to erratic, data-dependent stack behavior. Of course, if the stack is allowed to overflow, it will corrupt whatever lies above it (toward lower numbered addresses) in memory. And, as a result, unpredictable program behavior will likely occur.

To prevent this, some compiler writers implement a *stack probe* routine that, on entry to every function, checks to make sure that some reasonable amount of stack space remains. Since this necessarily involves a penalty in performance, it is implemented as a compile-time option. Typically, a program will compile with stack probes during testing, then when the programmer thinks the program is robust, he does the production compile without the safety net.

Since programs are usually compiled many times for just one production compile, stack probing should be a default that can be overridden with an **-NSP** switch, for example.

Two things are needed to implement stack probes—a tight little library routine for doing the checking, and a change to **dofunction()** to strategically place a call to the routine in each function. As it turns out, there is already such a function in the Small C library called **avail()**. When called with a *true* argument, the program aborts with an exit code of 1 if the stack has overflowed. While the needed functionality is there, this routine probably carries too much baggage for something to be called with every function call.

I suggest using **avail()** as a pattern for another function, **_probe()** for instance. This function would not be passed an argument, and would not return a value. It would, however, check the stack and abort with an exit code of say 3 (an unused Small C disaster code); this value would distinguish stack overflows from other insufficient memory problems.

Before leaving **avail()**, notice how it determines the value of SP. It declares a local character, then takes its address with the address operator (**&**). Since the character is the last thing on the stack, its address is the value of SP. Looking at

this, it is apparent that a more efficient way to do this would be to use the address of the argument **abort** instead. That would avoid the need to adjust SP to allocate the local variable. But it would also introduce an error of four bytes in the checking since the return address and original BP are "above" (lower numbered address) it on the stack. Is that error significant? Should **avail()** be revised?

Notice that you could declare a formal argument for **_probe()** even though no actual argument is passed. After all, the only thing you want is the address of where **_probe()** thinks the argument is. It does not actually have to be there.

With these modifications, a lean **_probe()** function can be written. For the ultimate in efficiency, however, you could write **_probe()** in assembly language. Then you could avoid the push of BP and its restoration on exit. You could also directly refer to SP. And, calling **abort()** with an argument of 3 is just a matter of

```
MOV AX,3
PUSH AX
CALL _abort
```

in assembly language.

Looking now at **dofunction()**, we must decide where to generate the call to **_probe()**. It could go just after generating **ENTER** which saves BP and then sets it for the new stack frame. At that point, if **-NSP** has not been specified, generate the call. Do you have a p-code for calling a function that is not in the symbol table? Should you put **_probe()** in the symbol table when **-NSP** is not given, or should you define another p-code for calling literal labels? (Note: If you should decide to define another p-code, don't forget to recompile all of the parts of the compiler that may be affected.)

Finally, since Small C allocates all locals in a block at one time, wouldn't it be better to wait until the locals are declared, then probe the stack? That way account is taken of the amount of local space used by the function. If that is done, then **statement()** should generate the call. The code in **statement()** that allocates locals, however, executes within very nested block. Should care be taken to ensure that **_probe()** is called only the first time in a function, or should you call probe every time a nested block is entered? Or perhaps, the first time for

sure, then subsequently only if additional locals are being allocated? How accurately should the stack be checked anyway?

Testing this revision, could be done with an infinite recursive function that calls **avail()** with each nesting level, in order to report progress by displaying the amount of free memory remaining. You can define a moderate size local array to gobble up stack space in reasonable chunks. Don't forget to test your revision to **ask()** which handles **-NSP**.

If done thoughtfully, this is probably a semester project. The decisions made along the way should be justified.

Long Integers

One of the most obvious enhancements to Small C is support for additional data types. Logically, the first step in that direction is to implement **long** integers. This would break the ice and set the stage for other more complex data types.

This step is a very major undertaking, affecting virtually every part of the compiler—one that I shall not, therefore, describe in detail.

Before starting this project, you should do *Consolidated Type Recognition* (described earlier). That will create a single function **dotype()** for parsing data type specifications. Having done that, you will need to revise that function to accept:

```
long
long int
unsigned long
unsigned long int
```

in addition to the types it already recognizes. For these cases it should return **LONG** or **ULONG** which should be defined with:

```
#define LONG  (BPW << 3)
#define ULONG (BPW << 3) + 1
```

in **CC.H**. This specifies the length (4) in the upper 14 bits and differentiates between the two in the lower two bits.

Next. **dodeclare()**, **dofunction()**, and **statement()** must be revised to accept the new data types and properly represent them in the symbol table.

A new p-code will be needed for defining longs with the **DD** (define double word) assembler directive.

The primary register will need to be extended to take in DX for the high-order 16 bits of long objects. It would consist of DX,AX together, in that order. Similarly, the secondary register must also be extended as CX,BX.

In memory, the low-order word should be stored first (with the lowest address). If the *Word Alignment* project has been performed, then it must be modified to recognize long types and to provide double-word alignment in their cases. The choice of double-word (instead of single-word) alignment would not improve performance on machines with 16-bit buses, but it would on those with 32-bit buses.

Function arguments and return values must be generalized to allow the passing and returning of double-word values. When passing a long argument **callfunc()** must generate:

```
PUSH DX
PUSH AX
```

instead of a single push. This can be handled easily by having **PUSH1** make use of its value to differentiate between single- and double-word pushes. Make use of the **?...?...?** device in the translation string. This suggests a parallel revision to **POP2** that will produce:

```
POP BX
POP CX
```

Dofunction() can no longer assume that all functions return integers. It must set the **TYPE** field in the symbol table to whatever type specification precedes the function name. This in turn will require a rewrite of **parse()** which now assumes it has a function definition when the type lead-in syntax is not seen by **dodeclare()**. **Doreturn()** must be revised to return the correct type of value, and to convert the return value to the correct type if necessary.

Until now, conversion of data from one type to another has not been a problem for Small C. Now it must be done, not only here but during expression analysis when binary operators are applied by **down2()**. See Kernighan and Ritchie [10] for details about the rules for data conversion.

Long objects must be fetched and stored like other objects, so **fetch()** and **store()** must take them into account. New p-codes will be needed for these purposes too.

This may seem like a lot of changes, but the largest one by far is the revision to the expression analyzer. Double-length constants should be supported, so **is[CV]** will have to become a sequence of two entries, and every reference to it must take that into account. **Number()** must be modified to recognize long constants and return their data types correctly.

Test() must be able to test double-length expression results. This also will require additional p-codes.

A major decision regarding revisions to p-codes must be made. How should precision be reflected in p-codes? The options appear to be (1) define additional p-codes for the new double word precision, (2) associate with each p-code and additional value that tells the precision, and (3) take advantage of the unused value that many of the affected p-codes already have. Option (1) is not good because it forces additional decisions to be made in the analyzer when a specific p-code must be chosen, and it multiplies the cases that the optimizer must deal with. Option (2) is bad because it requires 50% more space for the staging buffer. And, option (3) cannot be applied uniformly because some multi-precision p-codes already use their values for other purposes.

A compromise solution might be to use the unused high-order bits of the p-code itself. P-codes are treated as 16-bit values throughout the compiler, and the high order byte is always zero. That byte could be superimposed with the precision of the operation at the time the code is generated in the staging buffer. Care must be taken to avoid a conflict in the use of these same bits by the optimizer commands. This can be handled by using the high order nibble for the precision. The recognition phase of **peep()** could mask off those bits when it performs its matching operation. The optimizing phase must also be revised so that it will know what precision to apply to literal p-codes that it writes into the buffer.

Finally, there is the problem of how to translate p-codes according to their precision. Two approaches come to mind: (1) let the precision designate a substring (null separated) in the translation string, and (2) add the precision to the p-code proper to derive the subscript into **code[]**. Approach (1) has the disadvantage that translating p-codes will be slowed down by having to bypass unwanted substrings. Also, long strings will result, but that can be dealt with easily if the *Continued Character Strings* revision has been installed in the compiler. Approach (2) would require that you leave gaps in the p-code numbers (as they are assigned in **CC.H**) to account for the higher precision cases. The nuisance of this can be reduced by separating these p-codes from the others and placing them at the end of the list with appropriate comments.

These methods of dealing with the proliferation of p-codes need more thought, but perhaps these ideas will help guide you in the right direction. One last point to think of is that your solution should yield gracefully to additional precision data items. Having taken this step, they will certainly follow.

The analyzer will have to be scrutinized thoroughly to ensure that objects of different precision will be properly handled. Also, numeric constants larger than 32767 should be converted to double-length values, instead of being designated unsigned as they are now.

It should be obvious that provision must be made to pass the data type to **gen()** so that it can be properly associated with the p-code. Where does the data type originate? That's right, in the symbol table where **primary()** finds it. Fortunately, it looks as though **primary()** requires little revision, if any.

The place where unmatched data types will have to be reconciled is in **down2()**. Before applying a binary operator, **down2()** must generate code to convert the lower precision data type to the larger precision one if they differ. And it must make sure that the precision is properly passed on to **gen()**. In planning for the generation of type conversion code, look toward the possibility of doing *The Cast Operator* project below.

Down2() will have to scale values added to or subtracted from **long** addresses by four instead of two. This implies changes to **double()**. It must likewise generalize the way it scales the difference of two addresses.

Adding longs can be performed by generating in-line code like

```
ADD AX,BX
ADC DX,CX
```

where the second addition includes the carry from the first one. Likewise,

```
SUB AX,BX
SBB DX,CX
```

will handle **long** subtraction.

Multiplication and division are not so easy. In these cases, you will need to write library routines that are declared external only if they are actually used. Morgan [7] gives examples of multi-precision arithmetic routines from which you might get some ideas. Morgan's routines are more generalized than needed.

Finally, the run time library will need to be upgraded to deal with long integers. The library presently contains 12 functions that convert numbers between integer and ASCII string representations (signed decimal, unsigned decimal, hexadecimal, octal, and any base). For each of these, there should be a corresponding long integer equivalent. The new functions should be named **atol()**, **atolb()**, **dtol()**, **ltoa()**, **ltoab()**, **ltod()**, **ltoo()**, **ltou()**, **ltox()**, **otol()**, **utol()**, and **xtol()**. Having written these, they can be used in revising **printf()**, **fprintf()**, **scanf()**, and **fscanf()** to handle conversions between ASCII and long integers. This really involves only two functions, since the **f...()** version of these functions uses the same logic as the other version.

Having installed support for long integers, the stage will be set for other data types. At this point Small C will nearly be a mature compiler.

Obviously, this project should not be attempted unless you already have a comprehensive understanding of all parts of the compiler. This might do for a graduate level project.

The Cast Operator

The *Long Integers* project put the compiler in the business of performing data conversions and set the stage for the *cast* (or *coercion*) operator. A cast is really just a data conversion on demand. The idea is that by writing a data type in

parentheses before any (sub)expression, the result is "coerced" into the designated type. The hard part was done in the *Long Integers* project. It should only be necessary here to have **level13()** recognize

```
(Type)
```

as a unary operator. It should not, however, confuse the open parenthesis with a precedence altering parenthesis. This can be verified by the presence of a valid type specification following the opening parenthesis.

Since this small project depends on the *Long Integers* project, it could be used to supplement that project for extra credit or to help fulfill the requirements of a graduate project.

Floating Point Numbers

This project could also be viewed as a follow-on to the *Long Integers* project. But this one is hefty enough to serve as a separate assignment.

The first question to decide is what internal representation will be used for floating point numbers. With Intel's introduction of the 8087 family of numeric data processors (NDPs), any floating point compiler that targets the 8086 family processors should use the NDP if one is present. And, if one is not present, then the necessary operations should be simulated by library functions that provide comparable precision. I leave to you the task of obtaining specific information about the Intel NDPs and algorithms for manipulating floating point numbers. Suffice it here to say that doing the necessary research and writing the library routines for addition, subtraction, multiplication, and division is a significant project.

Having done that, you must then make the compiler recognize and properly handle floating point numbers. This is basically a matter of dealing with the same issues you encountered in the *Long Integers* project. The difference is that the major design decisions (about p-codes, etc.) have been made and a framework now exists for handling various data types. You will only be expanding on what is already in place.

Other Ideas

I could go on endlessly, describing other projects based on the Small C compiler. But at this point I will close by simply listing other ideas that come to mind.

1. Add support of structures and unions. For specific information on these, see Kernighan and Ritchie [10]. This, together with the preceding projects, would essentially elevate Small C to full C status.

2. Add **fseek()** and **ftell()** to the library. There are already nonstandard versions of these functions that work with integer offsets. But after implementing longs, these standard functions should be added. Use the existing **seek** and **tell** functions as patterns.

3. Enhance the Small C run-time library with **sprintf()** and **sscanf().** These versions of **printf()** and **scanf()** work with character strings instead of files. Pattern your functions after those in a professional package like Microsoft C.

4. Enhance the Small C run-time library with a set of functions that give full access to the facilities of operating system. Pattern your functions after those in a professional package like Microsoft C.

5. Enhance the Small C run-time library with a full set of math functions. This is an appropriate project to follow the implementation of floats. Pattern your functions after those in a professional package like Microsoft C.

6. Upgrade the sophistication of the Small C memory management routines so as to support the deallocation of memory blocks in any sequence. Reallocate embedded blocks of free memory before enlarging the heap. Combine adjacent blocks of released memory into a single block so that future requests will more likely find embedded space.

7. Have the compiler put the stack in a segment of its own so that the data segment can have all 64K bytes if it needs them. This is not a difficult thing to do. The start-up routine in **CALL.ASM** module will have to be revised. Also, the stack probe logic (above) will have to be revised.

8. Provide support for multiple memory models. Read the documentation for a professional C package, like Microsoft C, and follow their terminology and segment naming conventions. This should follow the *Long Integers* project since that project sets the stage for handling double-length pointers. Pointers of two

sizes must be handled—*near* pointers (single word) and far pointers (double word). Having done this, you can compile Small C in a larger memory model to open it up for further development.

9. Provide support for arrays of pointers. This can be done by fixing the symbol table to handle the additional information. But this is really only part of the general problem of having the compiler deal with multiple declaration modifiers. It should, for instance, be able to handle pointers to pointers. To do this properly, the declaration handling logic will have to be expanded into a kind of expression analysis, and symbol table entries must be tied to variable length chains of attributes.

10. Revise the preprocessor to support nesting of comments. This is a relatively simple, self-contained change. Take your ideas from the method the preprocessor already uses to handle **#if...** nesting.

11. Add support for local data initializers. Instead of always waiting for the first executable instruction in a block before allocating space for locals, when an initializer is found, allocate pending locals, then evaluate the initializing expression and push the result onto the stack as the current object. Whereas global initializers must be constant expressions, local initializers are not so restricted. Originally, C did not support initializing local arrays, yet current compilers tend to allow it. Is this something you want to support? Don't forget that the element values must be pushed from right to left so the first element will have the lowest address. Also, uninitialized elements should be set to zero.

12. Port the compiler to other operating systems on the same family of Intel processors. UNIX, XENIX, MINIX, and ZINU are possibilities. Essentially, this should be a matter of finding a suitable assembler, and adapting the run time library to the new operating system calls.

13. Port Small C to another processor. This involves all of the issues of the previous project plus reshaping the output code to the demands of a different processor. The assignment of the primary and secondary registers to physical registers must be decided. Problems will arise from different word lengths, different storage sequence for multi-precision data items, and so on.

APPENDIX A

The ASCII Character Set

dec	oct	hex			dec	oct	hex		dec	oct	hex	
0	0	0	^@	NUL	44	54	2C	,	88	130	58	X
1	1	1	^A	SOH	45	55	2D	-	89	131	59	Y
2	2	2	^B	STX	46	56	2E	.	90	132	5A	Z
3	3	3	^C	ETX	47	57	2F	/	91	133	5B	[
4	4	4	^D	EOT	48	60	30	0	92	134	5C	\
5	5	5	^E	ENQ	49	61	31	1	93	135	5D	]
6	6	6	^F	ACK	50	62	32	2	94	136	5E	^
7	7	7	^G	BEL	51	63	33	3	95	137	5F	_
8	10	8	^H	BS	52	64	34	4	96	140	60	`
9	11	9	^I	HT	53	65	35	5	97	141	61	a
10	12	A	^J	LF	54	66	36	6	98	142	62	b
11	13	B	^K	VT	55	67	37	7	99	143	63	c
12	14	C	^L	FF	56	70	38	8	100	144	64	d
13	15	D	^M	CR	57	71	39	9	101	145	65	e
14	16	E	^N	SO	58	72	3A	:	102	146	66	f
15	17	F	^O	SI	59	73	3B	;	103	147	67	g
16	20	10	^P	DLE	60	74	3C	<	104	150	68	h
17	21	11	^Q	DC1	61	75	3D	=	105	151	69	i
18	22	12	^R	DC2	62	76	3E	>	106	152	6A	j
19	23	13	^S	DC3	63	77	3F	?	107	153	6B	k
20	24	14	^T	DC4	64	100	40	@	108	154	6C	l
21	25	15	^U	NAK	65	101	41	A	109	155	6D	m
22	26	16	^V	SYN	66	102	42	B	110	156	6E	n
23	27	17	^W	ETB	67	103	43	C	111	157	6F	o
24	30	18	^X	CAN	68	104	44	D	112	160	70	p
25	31	19	^Y	EM	69	105	45	E	113	161	71	q
26	32	1A	^Z	SUB	70	106	46	F	114	162	72	r
27	33	1B	^[	ESC	71	107	47	G	115	163	73	s
28	34	1C	^\	FS	72	110	48	H	116	164	74	t
29	35	1D	^]	GS	73	111	49	I	117	165	75	u
30	36	1E	^^	RS	74	112	4A	J	118	166	76	v
31	37	1F	^_	US	75	113	4B	K	119	167	77	w
32	40	20		SP	76	114	4C	L	120	170	78	x
33	41	21	!		77	115	4D	M	121	171	79	y
34	42	22	"		78	116	4E	N	122	172	7A	z
35	43	23	#		79	117	4F	O	123	173	7B	{
36	44	24	$		80	120	50	P	124	174	7C	|
37	45	25	%		81	121	51	Q	125	175	7D	}
38	46	26	&		82	122	52	R	126	176	7E	~
39	47	27	'		83	123	53	S	127	177	7F	DEL
40	50	28	(		84	124	54	T				
41	51	29	)		85	125	55	U				
42	52	2A	*		86	126	56	V				
43	53	2B	+		87	127	57	W				

APPENDIX B

The 80x86-Family Processors

This appendix gives an overview of the Intel 8086 central processing unit (CPU) architecture. The 8086 is the base processor for a family of CPUs that execute the 8086 instruction set or supersets thereof. A chief difference between the CPUs in the family is the width of their data buses—8, 16, or 32 bits.

The original IBM PC used the 8088 CPU, which is a scaled down 8086 with an 8-bit data bus. The 8088 executes the same instruction set as the 8086. Higher performance CPUs based on the 8086 include the 80186, 80188, and the 80286. These have additional capabilities and execute a superset of the 8086 instructions.

The 80386 is a high speed 32-bit multi-mode CPU. In *real address mode*, it emulates an 8086 with some additional instructions. In *virtual 8086 mode*, it emulates an 8086 and can also switch quickly to *protected mode* to realize the full power of the 32-bit CPU. Switching back to *virtual 8086 mode* can also be done quickly.

Since the MS-DOS Small C compiler uses only the instructions of the basic 8086 CPU, it is compatible with all of these processors. The following material describes the basic architecture of the 80x86 family of CPUs.

Figure B-1 is a diagram of the 8086 central processing unit and its view of memory. Memory is simply an array of bytes. Each byte consists of eight bits and has a unique address. The first byte has an address of 0, the second 1, and so on through 1,048,575. In hexadecimal, the highest address is FFFFF, a 20-bit value.

The 80x86 family CPUs employ a memory segmentation scheme that permits 16-bit values to address more than 65,536 bytes. The CPU derives absolute addresses from two components—a 16-bit *segment address* and a 16-bit *offset*. The segment component comes from one of the four segment registers CS, DS,

SS, or ES. The offset component comes from one of the other registers or the instruction being executed. These are added together with the segment address shifted left four bits—that is, multiplied by 16. The result is the requisite 20-bit address.

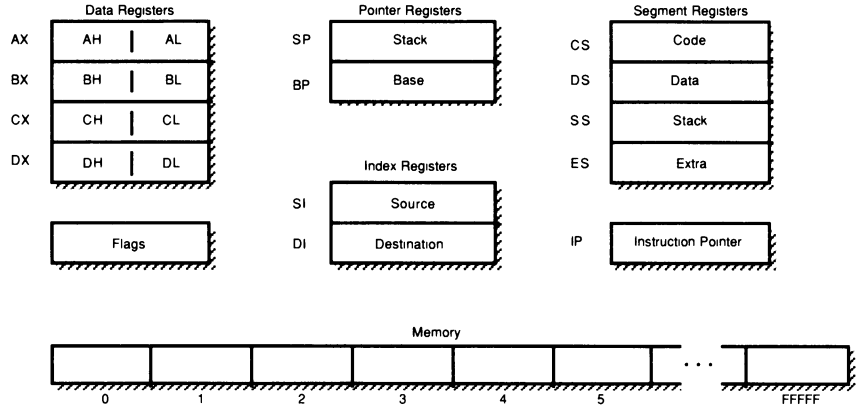


Figure B-1. 8086 CPU Architecture

A segment is a contiguous area of memory that begins on a *paragraph* boundary and has a maximum size of 65,536 bytes. Paragraph boundaries are addresses that are divisible by 16. Shifting the 16-bit paragraph address left four bits converts it to a memory address that must fall on a paragraph boundary.

The CPU is designed around the assumption that programs will usually consist of at least three segments—a *code segment* containing the program's instructions, a *data segment* containing global data, and a *stack segment* containing the program's stack. The *extra segment* register (ES) allows a program to have an additional data segment. Of course, a program could have more than one of each type of segment if it changed the values in the segment registers appropriately. On the other hand, all four segment registers could contain the same value, so that all of the program resides in a single segment.

When the CPU fetches an instruction for execution, the instruction is derived from the *code segment* (CS) and *instruction pointer* (IP) registers, as shown in Figure B-2.

When the CPU performs stack operations, it derives the stack address from the *stack segment* register (SS) and the *stack pointer* (SP) or the *base pointer* (BP). Figure B-3 illustrates the case for push and pop instructions. On the left is the situation before an operand is pushed onto the stack. On the right is the situation after the push. If a pop finds the situation on the right, it will leave it as on the left. The operand is not placed on the stack until the push instruction executes.

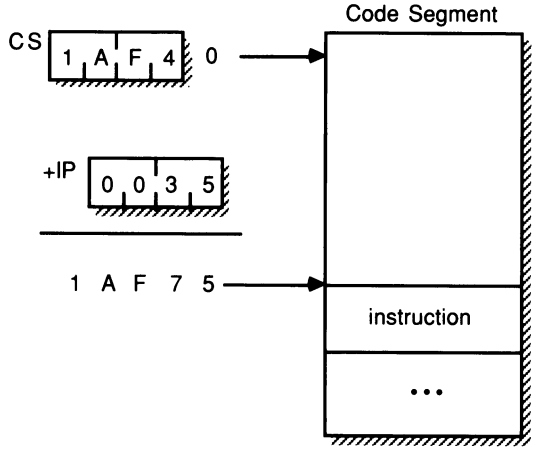


Figure B-2. 8086 Instruction Address

When the CPU makes an operand reference, it combines the *data segment* register (DS) with one or more offset values. In some cases the *extra segment* register (ES) is used instead. The default choice of DS for operand references can be overridden by an instruction prefix so that instructions can access any segment.

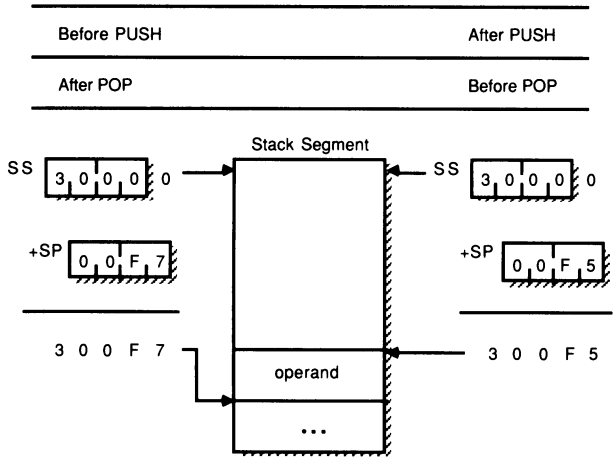


Figure B-3. 8086 Stack Address

8086 instructions access operands through any of six addressing modes. Table B-1 lists these modes.

Mode	Operand Location
Immediate	In the instruction.
Register	In a register.
Direct	In memory at the offset contained in the instruction.
Register Indirect	In memory at the offset contained in a register.
Based	In memory at the offset which is the sum of a Base Register, and the offset contained in the instruction.
Indexed	In memory at the offset, which is the sum of an Index Register, and the offset contained in the instruction.
Based and Indexed	In memory at the offset, which is the sum of a Base Register, an Index Register, and the offset contained in the instruction.

Table B-1. 8086 Operand Addressing Modes

In the *immediate* mode, the operand is a constant that is contained in the instruction. Immediate operands are fetched with the instruction, so they have no address references. In the *register* mode, the operand is contained in a register. The instruction only specifies which register contains the operand; therefore, no memory address is needed in that case either.

The remainder of the addressing modes require the CPU to compose a physical memory address for the operand. In the *direct* mode, the instruction contains an offset that is added to DS to derive the physical address of the operand. In the *register indirect* mode, a register contains an offset that is added to DS to produce the physical address of the operand. In the *based* mode, a base register is added to an offset contained in the instruction (a *displacement*) and to DS to produce the physical address of the operand. The instruction displacement is optional. In the *indexed* mode, an index register is added to a displacement in the instruction and to DS to produce the physical address of the operand. The instruction displacement is optional. In the *based and indexed* mode, a base register is added to an index register, a displacement in the instruction, and to DS to produce the physical address of the operand. The instruction displacement is optional. Figure B-4 illustrates an example of based and indexed addressing.

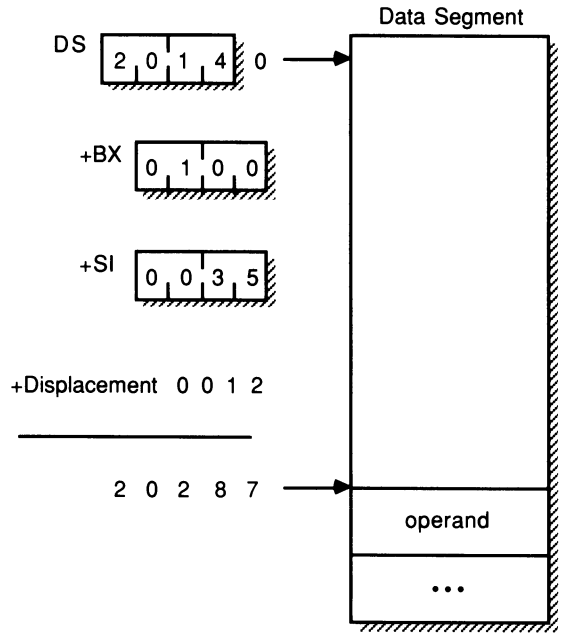


Figure B-4. Four Component Data Address

Many instructions can operate in any of the addressing modes. Instructions that have two operands are restricted as to which combinations of addressing modes are legal. Despite these restrictions, most instructions are symmetrical with respect to which is the source and which is the destination operand.

IP holds the offset of the next instruction to be fetched for execution. Normally, when an instruction is fetched, IP is incremented by the instruction's length, causing a sequential execution of instructions. However, jump, call, and return instructions break this sequence by placing a new value into IP.

SP and BP are used as offsets into the stack segment. BP establishes the base of a stack frame, whereas SP always points to the item at the top of the stack. BP is only changed by moving new values into it, whereas SP is also altered by push and pop instructions.

The index registers—*source index* (SI) and *destination index* (DI)—are used by certain string instructions as offsets into data segments. By default SI is associated with DS and DI with ES.

Sixteen-bit values (*words*) are stored in memory as consecutive bytes with the low-order byte first. Thirty-two-bit values (double words) are stored as consecutive 16-bit values with the low-order word first. Addresses that are a multiple of 2 and 4 are referred to as *word* and *double-word boundaries*. Depending on the width of the data bus, significant improvements in speed can be realized by placing word items on word boundaries, and double-word items on word or double-word boundaries.

Four data registers are included in the CPU. They are AX, BX, CX, and DX. Each of these 16-bit registers consists of a pair of 8-bit registers—a high register and a low register. Thus, AX consists of AH and AL, and so on. Instructions can reference the data registers as four 16-bit registers or eight 8-bit registers.

The flag register is a collection of condition flags which are set or cleared by conditions resulting from the execution of arithmetic and logical instructions. The flag bits can be tested individually. The 8086 CPU uses nine flag bits.

The *carry flag* (CF) is set by a carry into the high-order bit or a borrow from the high-order bit. The *auxiliary flag* (AF) is set by a carry out of the lowest nibble (4 bits) or a borrow into the lowest nibble. The *overflow flag* (OF) is set when an arithmetic overflow occurs. The *sign flag* (SF) is set when the high-order bit of the result is set to one. The *parity flag* (PF) is set when the result contains an even number of bits. The *zero flag* (ZF) is set when the result is zero.

Three of the flags control the CPU rather than reflect conditions resulting from an arithmetic or logical instruction. The *direction flag* (DF) determines the direction in which string operations are performed. When clear, string operations proceed from left to right, and vice versa. The *interrupt-enable flag* (IF) determines whether or not external (maskable) interrupts can be recognized. When clear, these interrupts are disabled, and vice versa. The *trap flag* (TF), when set, places the CPU in single step mode. In that mode, the CPU generates an interrupt after each instruction. This enables a debugger to receive control after each instruction.

APPENDIX C

Small C Compiler Listings

notice.h

```
/*
** NOTICE.H - Small C Signon Notice.
*/
#define VERSION "Small C, Version 2.2, Revision Level 112\n"
#define CRIGHT1 "Copyright 1982, 1983, 1985, 1988 J. E. Hendrix\n\n"
```

cc.h

```
/*
** CC.H - Symbol Definitions for Small-C compiler.
*/

/*
** machine dependent parameters
*/
#define BPW     2   /* bytes per word */
#define LBPW    1   /* log2(BPW) */
#define SBPC    1   /* stack bytes per character */
#define ERRCODE 7   /* op sys return code */

/*
** symbol table format
*/
#define IDENT    0
#define TYPE     1
#define CLASS    2
#define SIZE     3
#define OFFSET   5
#define NAME     7

#define SYMAVG  12
```

```
#define SYMMAX  16

/*
** symbol table parameters
*/
#define NUMLOCS   25
#define STARTLOC  symtab
#define ENDLOC    (symtab+NUMLOCS*SYMAVG)
#define NUMGLBS   200
#define STARTGLB  ENDLOC
#define ENDGLB    (ENDLOC+(NUMGLBS-1)*SYMMAX)
#define SYMTBSZ   3050  /* (NUMLOCS*SYMAVG + NUMGLBS*SYMMAX) */

/*
** system wide name size (for symbols)
*/
#define NAMESIZE 9
#define NAMEMAX  8

/*
** values for "IDENT"
*/
#define LABEL    0
#define VARIABLE 1
#define ARRAY    2
#define POINTER  3
#define FUNCTION 4

/*
** values for "TYPE"
**    high order 14 bits give length of object
**    low order 2 bits make type unique within length
*/
/*      LABEL   0 */
#define CHR     (  1 << 2)
#define INT     (BPW << 2)
#define UCHR   ((  1 << 2) + 1)
#define UINT   ((BPW << 2) + 1)
#define UNSIGNED             1

/*
** values for "CLASS"
*/
/*      LABEL     0 */
```

```
#define AUTOMATIC 1
#define STATIC    2
#define EXTERNAL  3
#define AUTOEXT   4

/*
** segment types
*/
#define DATASEG 1
#define CODESEG 2

/*
** "switch" table
*/
#define SWSIZ   (2*BPW)
#define SWTABSZ (90*SWSIZ)

/*
** "while" queue
*/
#define WQTABSZ  30
#define WQSIZ     3
#define WQMAX   (wq+WQTABSZ-WQSIZ)

/*
** field offsets in "while" queue
*/
#define WQSP    0
#define WQLOOP  1
#define WQEXIT  2

/*
** literal pool
*/
#define LITABSZ 2000
#define LITMAX  (LITABSZ-1)

/*
** input line
*/
#define LINEMAX  127
#define LINESIZE 128

/*
```

```
** entries in staging buffer
*/
#define STAGESIZE   200

/*
** macro (#define) pool
*/
#define MACNBR   300
#define MACNSIZE (MACNBR*(NAMESIZE+2))
#define MACNEND  (macn+MACNSIZE)
#define MACQSIZE (MACNBR*7)
#define MACMAX   (MACQSIZE-1)

/*
** statement types
*/
#define STIF      1
#define STWHILE   2
#define STRETURN  3
#define STBREAK   4
#define STCONT    5
#define STASM     6
#define STEXPR    7
#define STDO      8
#define STFOR     9
#define STSWITCH 10
#define STCASE   11
#define STDEF    12
#define STGOTO   13
#define STLABEL  14

/*
** p-code symbols
**
** legend:
**  1 = primary register (pr in comments)
**  2 = secondary register (sr in comments)
**  b = byte
**  f = jump on false condition
**  l = current literal pool label number
**  m = memory reference by label
**  n = numeric constant
**  p = indirect reference thru pointer in sr
**  r = repeated r times
```

```
**  s = stack frame reference
**  u = unsigned
**  w = word
**  _ (tail) = another p-code completes this one
*/

        /* compiler-generated */
#define ADD12     1   /* add sr to pr */
#define ADDSP     2   /* add to stack pointer */
#define AND12     3   /* AND sr to pr */
#define ANEG1     4   /* arith negate pr */
#define ARGCNTn   5   /* pass arg count to function */
#define ASL12     6   /* arith shift left sr by pr into pr */
#define ASR12     7   /* arith shift right sr by pr into pr */
#define CALL1     8   /* call function thru pr */
#define CALLm     9   /* call function directly */
#define BYTE_    10   /* define bytes (part 1) */
#define BYTEn    11   /* define byte of value n */
#define BYTErO   12   /* define r bytes of value 0 */
#define COM1     13   /* ones complement pr */
#define DBL1     14   /* double pr */
#define DBL2     15   /* double sr */
#define DIV12    16   /* div pr by sr */
#define DIV12u   17   /* div pr by sr unsigned */
#define ENTER    18   /* set stack frame on function entry */
#define EQ1Of    19   /* jump if (pr == 0) is false */
#define EQ12     20   /* set pr TRUE if (sr == pr) */
#define GE1Of    21   /* jump if (pr >= 0) is false */
#define GE12     22   /* set pr TRUE if (sr >= pr) */
#define GE12u    23   /* set pr TRUE if (sr >= pr) unsigned */
#define POINT1l  24   /* point pr to function's literal pool */
#define POINT1m  25   /* point pr to mem item thru label */
#define GETb1m   26   /* get byte into pr from mem thru label */
#define GETb1mu  27   /* get unsigned byte into pr from mem thru label */
#define GETb1p   28   /* get byte into pr from mem thru sr ptr */
#define GETb1pu  29   /* get unsigned byte into pr from mem thru sr ptr */
#define GETw1m   30   /* get word into pr from mem thru label */
#define GETw1n   31   /* get word of value n into pr */
#define GETw1p   32   /* get word into pr from mem thru sr ptr */
#define GETw2n   33   /* get word of value n into sr */
#define GT1Of    34   /* jump if (pr > 0) is false */
#define GT12     35   /* set pr TRUE if (sr > pr) */
#define GT12u    36   /* set pr TRUE if (sr > pr) unsigned */
#define WORD_    37   /* define word (part 1) */
```

```
#define WORDn    38   /* define word of value n */
#define WORDr0   39   /* define r words of value 0 */
#define JMPm     40   /* jump to label */
#define LABm     41   /* define label m */
#define LE10f    42   /* jump if (pr <= 0) is false */
#define LE12     43   /* set pr TRUE if (sr <= pr) */
#define LE12u    44   /* set pr TRUE if (sr <= pr) unsigned */
#define LNEG1    45   /* logical negate pr */
#define LT10f    46   /* jump if (pr < 0) is false */
#define LT12     47   /* set pr TRUE if (sr < pr) */
#define LT12u    48   /* set pr TRUE if (sr < pr) unsigned */
#define MOD12    49   /* modulo pr by sr */
#define MOD12u   50   /* modulo pr by sr unsigned */
#define MOVE21   51   /* move pr to sr */
#define MUL12    52   /* multiply pr by sr */
#define MUL12u   53   /* multiply pr by sr unsigned */
#define NE10f    54   /* jump if (pr != 0) is false */
#define NE12     55   /* set pr TRUE if (sr != pr) */
#define NEARm    56   /* define near pointer thru label */
#define OR12     57   /* OR sr onto pr */
#define POINT1s  58   /* point pr to stack item */
#define POP2     59   /* pop stack into sr */
#define PUSH1    60   /* push pr onto stack */
#define PUTbm1   61   /* put pr byte in mem thru label */
#define PUTbp1   62   /* put pr byte in mem thru sr ptr */
#define PUTwm1   63   /* put pr word in mem thru label */
#define PUTwp1   64   /* put pr word in mem thru sr ptr */
#define rDEC1    65   /* dec pr (may repeat) */
#define REFm     66   /* finish instruction with label */
#define RETURN   67   /* restore stack and return */
#define rINC1    68   /* inc pr (may repeat) */
#define SUB12    69   /* sub sr from pr */
#define SWAP12   70   /* swap pr and sr */
#define SWAP1s   71   /* swap pr and top of stack */
#define SWITCH   72   /* find switch case */
#define XOR12    73   /* XOR pr with sr */

        /* optimizer-generated */
#define ADD1n    74   /* add n to pr */
#define ADD21    75   /* add pr to sr */
#define ADD2n    76   /* add immediate to sr */
#define ADDbpn   77   /* add n to mem byte thru sr ptr */
#define ADDwpn   78   /* add n to mem word thru sr ptr */
#define ADDm_    79   /* add n to mem byte/word thru label (part 1) */
```

```
#define COMMAn   80   /* finish instruction with ,n */
#define DECbp    81   /* dec mem byte thru sr ptr */
#define DECwp    82   /* dec mem word thru sr ptr */
#define POINT2m  83   /* point sr to mem thru label */
#define POINT2m_ 84   /* point sr to mem thru label (part 1) */
#define GETb1s   85   /* get byte into pr from stack */
#define GETb1su  86   /* get unsigned byte into pr from stack */
#define GETw1m_  87   /* get word into pr from mem thru label (part 1) */
#define GETw1s   88   /* get word into pr from stack */
#define GETw2m   89   /* get word into sr from mem (label) */
#define GETw2p   90   /* get word into sr thru sr ptr */
#define GETw2s   91   /* get word into sr from stack */
#define INCbp    92   /* inc byte in mem thru sr ptr */
#define INCwp    93   /* inc word in mem thru sr ptr */
#define PLUSn    94   /* finish instruction with +n */
#define POINT2s  95   /* point sr to stack */
#define PUSH2    96   /* push sr */
#define PUSHm    97   /* push word from mem thru label */
#define PUSHp    98   /* push word from mem thru sr ptr */
#define PUSHs    99   /* push word from stack */
#define PUT_m_  100   /* put byte/word into mem thru label (part 1) */
#define rDEC2   101   /* dec sr (may repeat) */
#define rINC2   102   /* inc sr (may repeat) */
#define SUB_m_  103   /* sub from mem byte/word thru label (part 1) */
#define SUB1n   104   /* sub n from pr */
#define SUBbpn  105   /* sub n from mem byte thru sr ptr */
#define SUBwpn  106   /* sub n from mem word thru sr ptr */

#define PCODES  107   /* size of code[] */
```

stdio.h

```
/*
** STDIO.H - Standard Small C Definitions.
*/
#define stdin    0  /* file descriptor for standard input file */
#define stdout   1  /* file descriptor for standard output file */
#define stderr   2  /* file descriptor for standard error file */
#define stdaux   3  /* file descriptor for standard auxiliary port */
#define stdprn   4  /* file descriptor for standard printer */
#define FILE  char  /* supports "FILE *fp;" declarations */
#define ERR   (-2)  /* return value for errors */
#define EOF   (-1)  /* return value for end-of-file */
```

```
#define YES      1  /* true */
#define NO       0  /* false */
#define NULL     0  /* zero */
#define CR      13  /* ASCII carriage return */
#define LF      10  /* ASCII line feed */
#define BELL     7  /* ASCII bell */
#define SPACE  ' '  /* ASCII space */
#define NEWLINE LF  /* Small C newline character */
```

cc1.c

```
/*
** Small-C Compiler — Part 1 —  Top End.
** Copyright 1982, 1983, 1985, 1988 J. E. Hendrix
** All rights reserved.
*/

#include <stdio.h>
#include "notice.h"
#include "cc.h"

/*
** miscellaneous storage
*/
int
  nogo,     /* disable goto statements? */
  noloc,    /* disable block locals? */
  opindex,  /* index to matched operator */
  opsize,   /* size of operator in characters */
  swactive, /* inside a switch? */
  swdefault,/* default label #, else 0 */
 *swnext,   /* address of next entry */
 *swend,    /* address of last entry */
 *stage,    /* staging buffer address */
 *wq,       /* while queue */
  argcs,    /* static argc */
 *argvs,    /* static argv */
 *wqptr,    /* ptr to next entry */
  litptr,   /* ptr to next entry */
  macptr,   /* macro buffer index */
  pptr,     /* ptr to parsing buffer */
  ch,       /* current character of input line */
  nch,      /* next character of input line */
```

```
  declared, /* # of local bytes to declare, -1 when declared */
  iflevel,  /* #if... nest level */
  skiplevel,/* level at which #if... skipping started */
  nxtlab,   /* next avail label # */
  litlab,   /* label # assigned to literal pool */
  csp,      /* compiler relative stk ptr */
  argstk,   /* function arg sp */
  argtop,   /* highest formal argument offset */
  ncmp,     /* # open compound statements */
  errflag,  /* true after 1st error in statement */
  eof,      /* true on final input eof */
  output,   /* fd for output file */
  files,    /* true if file list specified on cmd line */
  filearg,  /* cur file arg index */
  input   = EOF, /* fd for input file */
  input2  = EOF, /* fd for "#include" file */
  usexpr  = YES, /* true if value of expression is used */
  ccode   = YES, /* true while parsing C code */
 *snext,    /* next addr in stage */
 *stail,    /* last addr of data in stage */
 *slast,    /* last addr in stage */
  listfp,   /* file pointer to list device */
  laststt,  /* last parsed statement type */
  oldseg;   /* current segment (0, DATASEG, CODESEG) */

char
  optimize, /* optimize output of staging buffer? */
  alarm,    /* audible alarm on errors? */
  monitor,  /* monitor function headers? */
  pause,    /* pause for operator on errors? */
 *symtab,   /* symbol table */
 *litq,     /* literal pool */
 *macn,     /* macro name buffer */
 *macq,     /* macro string buffer */
 *pline,    /* parsing buffer */
 *mline,    /* macro buffer */
 *line,     /* ptr to pline or mline */
 *lptr,     /* ptr to current character in "line" */
 *glbptr,   /* global symbol table */
 *locptr,   /* next local symbol table entry */
  quote[2] = {'"'}, /* literal string for '"' */
 *cptr,     /* work ptrs to any char buffer */
 *cptr2,
 *cptr3,
```

```
  msname[NAMESIZE],   /* macro symbol name */
  ssname[NAMESIZE];   /* static symbol name */

int op[16] = {   /* p-codes of signed binary operators */
  OR12,                         /* level5 */
  XOR12,                        /* level6 */
  AND12,                        /* level7 */
  EQ12,   NE12,                 /* level8 */
  LE12,   GE12,  LT12,  GT12,   /* level9 */
  ASR12,  ASL12,                /* level10 */
  ADD12,  SUB12,                /* level11 */
  MUL12, DIV12, MOD12           /* level12 */
  };

int op2[16] = {  /* p-codes of unsigned binary operators */
  OR12,                         /* level5 */
  XOR12,                        /* level6 */
  AND12,                        /* level7 */
  EQ12,   NE12,                 /* level8 */
  LE12u,  GE12u, LT12u, GT12u,  /* level9 */
  ASR12,  ASL12,                /* level10 */
  ADD12,  SUB12,                /* level11 */
  MUL12u, DIV12u, MOD12u        /* level12 */
  };

/*
** execution begins here
*/
main(argc, argv) int argc, *argv; {
  fputs(VERSION, stderr);
  fputs(CRIGHT1, stderr);
  argcs   = argc;
  argvs   = argv;
  swnext  = calloc(SWTABSZ, 1);
  swend   = swnext+(SWTABSZ-SWSIZ);
  stage   = calloc(STAGESIZE, 2*BPW);
  wqptr   =
  wq      = calloc(WQTABSZ, BPW);
  litq    = calloc(LITABSZ, 1);
  macn    = calloc(MACNSIZE, 1);
  macq    = calloc(MACQSIZE, 1);
  pline   = calloc(LINESIZE, 1);
  mline   = calloc(LINESIZE, 1);
  slast   = stage+(STAGESIZE*2*BPW);
```

```
  symtab  = calloc((NUMLOCS*SYMAVG + NUMGLBS*SYMMAX), 1);
  locptr  = STARTLOC;
  glbptr  = STARTGLB;

  ask();          /* get user options */
  openfile();     /* and initial input file */
  preprocess();   /* fetch first line */
  header();       /* intro code */
  setcodes();     /* initialize code pointer array */
  parse();        /* process ALL input */
  trailer();      /* follow-up code */
  fclose(output); /* explicitly close output */
  }

/******************* high level parsing ******************/

/*
** process all input text
**
** At this level, only static declarations,
**      defines, includes and function
**      definitions are legal...
*/
parse() {
  while (eof == 0) {
    if     (amatch("extern", 6)) dodeclare(EXTERNAL);
    else if(dodeclare(STATIC))   ;
    else if( match("#asm"))      doasm();
    else if( match("#include"))  doinclude();
    else if( match("#define"))   dodefine();
    else                         dofunction();
    blanks();                 /* force eof if pending */
    }
  }

/*
** test for global declarations
*/
dodeclare(class) int class; {
  if     (amatch("char",     4))  declglb(CHR,  class);
  else if(amatch("unsigned", 8)) {
    if   (amatch("char",     4))  declglb(UCHR, class);
    else {amatch("int",      3);  declglb(UINT, class);}
    }
```

```
  else if(amatch("int",      3)
       || class == EXTERNAL)      declglb(INT,  class);
  else return 0;
  ns();
  return 1;
  }

/*
** declare a static variable
*/
declglb(type, class)  int type, class; {
  int id, dim;
  while(1) {
    if(endst()) return;  /* do line */
    if(match("*"))       {id = POINTER;  dim = 0;}
    else                 {id = VARIABLE; dim = 1;}
    if(symname(ssname) == 0) illname();
    if(findglb(ssname)) multidef(ssname);
    if(id == VARIABLE) {
      if     (match("("))  {id = FUNCTION; need(")");}
      else if(match("["))  {id = ARRAY; dim = needsub();}
      }
    if     (class == EXTERNAL) external(ssname, type >> 2, id);
    else if(   id != FUNCTION) initials(type >> 2, id, dim);
    if(id == POINTER)
         addsym(ssname, id, type, BPW, 0, &glbptr, class);
    else addsym(ssname, id, type, dim * (type >> 2), 0, &glbptr, class);
    if(match(",") == 0) return;
    }
  }

/*
** initialize global objects
*/
initials(size, ident, dim) int size, ident, dim; {
  int savedim;
  litptr = 0;
  if(dim == 0) dim = -1;          /* *... or ...[] */
  savedim = dim;
  public(ident);
  if(match("=")) {
    if(match("{")) {
      while(dim) {
        init(size, ident, &dim);
```

```
        if(match(",") == 0) break;
        }
      need("}");
      }
    else init(size, ident, &dim);
    }
  if(savedim == -1 && dim == -1) {
    if(ident == ARRAY) error("need array size");
    stowlit(0, size = BPW);
    }
  dumplits(size);
  dumpzero(size, dim);           /* only if dim > 0 */
  }

/*
** evaluate one initializer
*/
init(size, ident, dim) int size, ident, *dim; {
  int value;
  if(string(&value)) {
    if(ident == VARIABLE || size != 1)
      error("must assign to char pointer or char array");
    *dim -= (litptr - value);
    if(ident == POINTER) point();
    }
  else if(constexpr(&value)) {
    if(ident == POINTER) error("cannot assign to pointer");
    stowlit(value, size);
    *dim -= 1;
    }
  }

/*
** get required array size
*/
needsub()  {
  int val;
  if(match("]")) return 0; /* null size */
  if(constexpr(&val) == 0) val = 1;
  if(val < 0) {
    error("negative size illegal");
    val = -val;
    }
  need("]");               /* force single dimension */
```

```
  return val;              /* and return size */
  }

/*
** open an include file
*/
doinclude() {
  int i; char str[30];
  blanks();       /* skip over to name */
  if(*lptr == '"' || *lptr == '<') ++lptr;
  i = 0;
  while(lptr[i]
     && lptr[i] != '"'
     && lptr[i] != '>'
     && lptr[i] != '\n') {
    str[i] = lptr[i];
    ++i;
    }
  str[i] = NULL;
  if((input2 = fopen(str,"r")) == NULL) {
    input2 = EOF;
    error("open failure on include file");
    }
  kill();   /* make next read come from new file (if open) */
  }

/*
** define a macro symbol
*/
dodefine() {
  int k;
  if(symname(msname) == 0) {
    illname();
    kill();
    return;
    }
  k = 0;
  if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0) == 0) {
    if(cptr2 = cptr)
      while(*cptr2++ = msname[k++]) ;
    else {
      error("macro name table full");
      return;
      }
```

```
    }
  putint(macptr, cptr+NAMESIZE, 2);
  while(white()) gch();
  while(putmac(gch()));
  if(macptr >= MACMAX) {
    error("macro string queue full");
    abort(ERRCODE);
    }
  }

putmac(c)  char c; {
  macq[macptr] = c;
  if(macptr < MACMAX) ++macptr;
  return c;
  }

/*
** begin a function
**
** called from "parse" and tries to make a function
** out of the following text
*/
dofunction()  {
  char *ptr;
  nogo   =                      /* enable goto statements */
  noloc  =                      /* enable block-local declarations */
  lastst =                      /* no statement yet */
  litptr = 0;                   /* clear lit pool */
  litlab = getlabel();          /* label next lit pool */
  locptr = STARTLOC;            /* clear local variables */
  if(match("void")) blanks();   /* skip "void" & locate header */
  if(monitor) lout(line, stderr);
  if(symname(ssname) == 0) {
    error("illegal function or declaration");
    errflag = 0;
    kill();                     /* invalidate line */
    return;
    }
  if(ptr = findglb(ssname)) {   /* already in symbol table? */
    if(ptr[CLASS] == AUTOEXT)
         ptr[CLASS] = STATIC;
    else multidef(ssname);
    }
  else addsym(ssname, FUNCTION, INT, 0, 0, &glbptr, STATIC);
```

```
  public(FUNCTION);
  argstk = 0;                    /* init arg count */
  if(match("(") == 0) error("no open paren");
  while(match(")") == 0) {       /* then count args */
    if(symname(ssname)) {
      if(findloc(ssname)) multidef(ssname);
      else {
        addsym(ssname, 0, 0, 0, argstk, &locptr, AUTOMATIC);
        argstk += BPW;
        }
      }
    else {
      error("illegal argument name");
      skip();
      }
    blanks();
    if(streq(lptr,")") == 0 && match(",") == 0)
      error("no comma");
    if(endst()) break;
    }
  csp = 0;                       /* preset stack ptr */
  argtop = argstk+BPW;           /* account for the pushed BP */
  while(argstk) {
    if     (amatch("char",     4)) {doargs(CHR);  ns();}
    else if(amatch("int",      3)) {doargs(INT);  ns();}
    else if(amatch("unsigned", 8)) {
      if   (amatch("char", 4))     {doargs(UCHR); ns();}
      else {amatch("int", 3);       doargs(UINT); ns();}
      }
    else {error("wrong number of arguments"); break;}
    }
  gen(ENTER, 0);
  statement();
  if(laststst != STRETURN && laststst != STGOTO)
    gen(RETURN, 0);
  if(litptr) {
    toseg(DATASEG);
    gen(REFm, litlab);
    dumplits(1);                 /* dump literals */
    }
  }

/*
** declare argument types
```

```
*/
doargs(type) int type; {
  int id, sz;
  char c, *ptr;
  while(1) {
    if(argstk == 0) return;           /* no arguments */
    if(decl(type, POINTER, &id, &sz)) {
      if(ptr = findloc(ssname)) {
        ptr[IDENT] = id;
        ptr[TYPE]  = type;
        putint(sz, ptr+SIZE, 2);
        putint(argtop-getint(ptr+OFFSET, 2), ptr+OFFSET, 2);
        }
      else error("not an argument");
      }
    argstk = argstk - BPW;            /* cnt down */
    if(endst()) return;
    if(match(",") == 0) error("no comma");
    }
  }

/*
** parse next local or argument declaration
*/
decl(type, aid, id, sz) int type, aid, *id, *sz; {
  int n, p;
  if(match("(")) p = 1;
  else           p = 0;
  if(match("*"))        {*id = POINTER;  *sz  = BPW;}
  else                  {*id = VARIABLE; *sz  = type >> 2;}
  if((n = symname(ssname)) == 0) illname();
  if(p && match(")")) ;
  if(match("(")) {
    if(!p || *id != POINTER) error("try (*...)()");
    need(")");
    }
  else if(*id == VARIABLE && match("[")) {
    *id = aid;
    if((*sz *= needsub()) == 0) {
      if(aid == ARRAY) error("need array size");
      *sz  = BPW;      /* size of pointer argument */
      }
    }
  return n;
```

```
  }

/******************** start 2nd level parsing ******************/

/*
** statement parser
*/
statement() {
  if(ch == 0 && eof) return;
  else if(amatch("char",     4)) {declloc(CHR);    ns();}
  else if(amatch("int",      3)) {declloc(INT);    ns();}
  else if(amatch("unsigned", 8)) {
    if   (amatch("char",     4)) {declloc(UCHR);   ns();}
    else {amatch("int",      3);  declloc(UINT);   ns();}
    }
  else {
    if(declared >= 0) {
      if(ncmp > 1) nogo = declared;   /* disable goto */
      gen(ADDSP, csp - declared);
      declared = -1;
      }
    if(match("{"))                compound();
    else if(amatch("if",       2)) {doif();          laststst = STIF;}
    else if(amatch("while",    5)) {dowhile();       laststst = STWHILE;}
    else if(amatch("do",       2)) {dodo();          laststst = STDO;}
    else if(amatch("for",      3)) {dofor();         laststst = STFOR;}
    else if(amatch("switch",   6)) {doswitch();      laststst = STSWITCH;}
    else if(amatch("case",     4)) {docase();        laststst = STCASE;}
    else if(amatch("default",  7)) {dodefault();     laststst = STDEF;}
    else if(amatch("goto",     4)) {dogoto();        laststst = STGOTO;}
    else if(dolabel())                               laststst = STLABEL;
    else if(amatch("return",   6)) {doreturn(); ns(); laststst = STRETURN;}
    else if(amatch("break",    5)) {dobreak();  ns(); laststst = STBREAK;}
    else if(amatch("continue", 8)) {docont();   ns(); laststst = STCONT;}
    else if(match(";"))            errflag = 0;
    else if(match("#asm"))         {doasm();          laststst = STASM;}
    else                           {doexpr(NO); ns(); laststst = STEXPR;}
    }
  return laststst;
  }

/*
** declare local variables
*/
```

```
declloc(type)  int type;  {
  int id, sz;
  if(swactive)     error("not allowed in switch");
  if(noloc)        error("not allowed with goto");
  if(declared < 0) error("must declare first in block");
  while(1) {
    if(endst()) return;
    decl(type, ARRAY, &id, &sz);
    declared += sz;
    addsym(ssname, id, type,  sz, csp - declared, &locptr, AUTOMATIC);
    if(match(",") == 0) return;
    }
  }

compound()  {
  int savcsp;
  char *savloc;
  savcsp = csp;
  savloc = locptr;
  declared = 0;           /* may now declare local variables */
  ++ncmp;                 /* new level open */
  while (match("}") == 0)
    if(eof) {
      error("no final }");
      break;
      }
    else statement();     /* do one */
  if(-ncmp               /* close current level */
  && laststst != STRETURN
  && laststst != STGOTO)
    gen(ADDSP, savcsp);   /* delete local variable space */
  cptr = savloc;          /* retain labels */
  while(cptr < locptr) {
    cptr2 = nextsym(cptr);
    if(cptr[IDENT] == LABEL) {
      while(cptr < cptr2) *savloc++ = *cptr++;
      }
    else cptr = cptr2;
    }
  locptr = savloc;        /* delete local symbols */
  declared = -1;          /* may not declare variables */
  }

doif()  {
```

```
  int flab1, flab2;
  test(flab1 = getlabel(), YES);  /* get expr, and branch false */
  statement();                    /* if true, do a statement */
  if(amatch("else", 4) == 0) {    /* if...else ? */
    /* simple "if"...print false label */
    gen(LABm, flab1);
    return;                       /* and exit */
    }
  flab2 = getlabel();
  if(laststst != STRETURN && laststst != STGOTO)
    gen(JMPm, flab2);
  gen(LABm, flab1);    /* print false label */
  statement();         /* and do "else" clause */
  gen(LABm, flab2);    /* print true label */
  }

dowhile()  {
  int wq[4];              /* allocate local queue */
  addwhile(wq);           /* add entry to queue for "break" */
  gen(LABm, wq[WQLOOP]);  /* loop label */
  test(wq[WQEXIT], YES);  /* see if true */
  statement();            /* if so, do a statement */
  gen(JMPm, wq[WQLOOP]);  /* loop to label */
  gen(LABm, wq[WQEXIT]);  /* exit label */
  delwhile();             /* delete queue entry */
  }

dodo() {
  int wq[4];
  addwhile(wq);
  gen(LABm, wq[WQLOOP]);
  statement();
  need("while");
  test(wq[WQEXIT], YES);
  gen(JMPm, wq[WQLOOP]);
  gen(LABm, wq[WQEXIT]);
  delwhile();
  ns();
  }

dofor() {
  int wq[4], lab1, lab2;
  addwhile(wq);
  lab1 = getlabel();
```

```
  lab2 = getlabel();
  need("(");
  if(match(";") == 0) {
    doexpr(NO);           /* expr 1 */
    ns();
    }
  gen(LABm, lab1);
  if(match(";") == 0) {
    test(wq[WQEXIT], NO); /* expr 2 */
    ns();
    }
  gen(JMPm, lab2);
  gen(LABm, wq[WQLOOP]);
  if(match(")") == 0) {
    doexpr(NO);           /* expr 3 */
    need(")");
    }
  gen(JMPm, lab1);
  gen(LABm, lab2);
  statement();
  gen(JMPm, wq[WQLOOP]);
  gen(LABm, wq[WQEXIT]);
  delwhile();
  }

doswitch() {
  int wq[4], endlab, swact, swdef, *swnex, *swptr;
  swact = swactive;
  swdef = swdefault;
  swnex = swptr = swnext;
  addwhile(wq);
  *(wqptr + WQLOOP - WQSIZ) = 0;
  need("(");
  doexpr(YES);                /* evaluate switch expression */
  need(")");
  swdefault = 0;
  swactive = 1;
  gen(JMPm, endlab = getlabel());
  statement();                /* cases, etc. */
  gen(JMPm, wq[WQEXIT]);
  gen(LABm, endlab);
  gen(SWITCH, 0);             /* match cases */
  while(swptr < swnext) {
    gen(NEARm, *swptr++);
```

```
    gen(WORDn,  *swptr++);    /* case value */
    }
  gen(WORDn, 0);
  if(swdefault) gen(JMPm, swdefault);
  gen(LABm, wq[WQEXIT]);
  delwhile();
  swnext    = swnex;
  swdefault = swdef;
  swactive  = swact;
  }

docase() {
  if(swactive == 0) error("not in switch");
  if(swnext > swend) {
    error("too many cases");
    return;
    }
  gen(LABm, *swnext++ = getlabel());
  constexpr(swnext++);
  need(":");
  }

dodefault() {
  if(swactive) {
    if(swdefault) error("multiple defaults");
    }
  else error("not in switch");
  need(":");
  gen(LABm, swdefault = getlabel());
  }

dogoto() {
  if(nogo > 0) error("not allowed with block-locals");
  else noloc = 1;
  if(symname(ssname)) gen(JMPm, addlabel());
  else error("bad label");
  ns();
  }

dolabel() {
  char *savelptr;
  blanks();
  savelptr = lptr;
  if(symname(ssname)) {
```

```
    if(gch() == ':') {
      gen(LABm, addlabel());
      return 1;
      }
    else bump(savelptr-lptr);
    }
  return 0;
  }

addlabel()  {
  if(cptr = findloc(ssname)) {
    if(cptr[IDENT] != LABEL) error("not a label");
    }
  else cptr = addsym(ssname, LABEL, LABEL, 0, getlabel(), &locptr, LABEL);
  return (getint(cptr+OFFSET, 2));
  }

doreturn()  {
  int savcsp;
  if(endst() == 0) doexpr(YES);
  savcsp = csp;
  gen(RETURN, 0);
  csp = savcsp;
  }

dobreak()  {
  int *ptr;
  if((ptr = readwhile(wqptr)) == 0) return;
  gen(ADDSP, ptr[WQSP]);
  gen(JMPm, ptr[WQEXIT]);
  }

docont()  {
  int *ptr;
  ptr = wqptr;
  while (1) {
    if((ptr = readwhile(ptr)) == 0) return;
    if(ptr[WQLOOP]) break;
    }
  gen(ADDSP, ptr[WQSP]);
  gen(JMPm, ptr[WQLOOP]);
  }

doasm()  {
```

```
  ccode = 0;            /* mark mode as "asm" */
  while (1) {
    inline();
    if(match("#endasm")) break;
    if(eof)break;
    fputs(line, output);
    }
  kill();
  ccode = 1;
  }

doexpr(use) int use; {
  int const, val;
  int *before, *start;
  usexpr = use;         /* tell isfree() whether expr value is used */
  while(1) {
    setstage(&before, &start);
    expression(&const, &val);
    clearstage(before, start);
    if(ch != ',') break;
    bump(1);
    }
  usexpr = YES;         /* return to normal value */
  }

/******************** miscellaneous functions ******************/

/*
** get run options
*/
ask() {
  int i;
  i = listfp = nxtlab = 0;
  output = stdout;
  optimize = YES;
  alarm = monitor = pause = NO;
  line = mline;
  while(getarg(++i, line, LINESIZE, argcs, argvs) != EOF) {
    if(line[0] != '-') continue;
    if(toupper(line[1]) == 'L'
    && isdigit(line[2])
    && line[3] <= ' ') {
      listfp = line[2]-'0';
      continue;
```

```
      }
    if(toupper(line[1]) == 'N'
    && toupper(line[2]) == 'O'
    && line[3] <= ' ') {
      optimize = NO;
      continue;
      }
    if(line[2] <= ' ') {
      if(toupper(line[1]) == 'A') {alarm   = YES; continue;}
      if(toupper(line[1]) == 'M') {monitor = YES; continue;}
      if(toupper(line[1]) == 'P') {pause   = YES; continue;}
      }
    fputs("usage: cc [file]... [-m] [-a] [-p] [-l#] [-no]\n", stderr);
    abort(ERRCODE);
    }
  }

/*
** input and output file opens
*/
openfile() {        /* entire function revised */
  char outfn[15];
  int i, j, ext;
  input = EOF;
  while(getarg(++filearg, pline, LINESIZE, argcs, argvs) != EOF) {
    if(pline[0] == '-') continue;
    ext = NO;
    i = -1;
    j = 0;
    while(pline[++i]) {
      if(pline[i] == '.') {
        ext = YES;
        break;
        }
      if(j < 10) outfn[j++] = pline[i];
      }
    if(!ext) strcpy(pline + i, ".C");
    input = mustopen(pline, "r");
    if(!files && iscons(stdout)) {
      strcpy(outfn + j, ".ASM");
      output = mustopen(outfn, "w");
      }
    files = YES;
    kill();
```

```
    return;
    }
  if(files++) eof = YES;
  else input = stdin;
  kill();
  }

/*
** open a file with error checking
*/
mustopen(fn, mode) char *fn, *mode; {
  int fd;
  if(fd = fopen(fn, mode)) return fd;
  fputs("open error on ", stderr);
  lout(fn, stderr);
  abort(ERRCODE);
  }
```

cc2.c

```
/*
** Small-C Compiler - Part 2 - Front End and Miscellaneous.
** Copyright 1982, 1983, 1985, 1988 J. E. Hendrix
** All rights reserved.
*/

#include <stdio.h>
#include "cc.h"

extern char
 *symtab, *macn, *macq, *pline, *mline,  optimize,
  alarm, *glbptr, *line, *lptr, *cptr, *cptr2,  *cptr3,
 *locptr, msname[NAMESIZE],  pause,  quote[2];

extern int
  *wq,  ccode,  ch,  csp,  eof,  errflag,  iflevel,
  input,  input2,  listfp,  macptr,  nch,
  nxtlab,  op[16],  opindex,  opsize,  output,  pptr,
  skiplevel,  *wqptr;

/********************** input functions **********************/

preprocess() {
```

```
int k;
char c;
if(ccode) {
  line = mline;
  ifline();
  if(eof) return;
  }
else {
  inline();
  return;
  }
pptr = -1;
while(ch != NEWLINE && ch) {
  if(white()) {
    keepch(' ');
    while(white()) gch();
    }
  else if(ch == '"') {
    keepch(ch);
    gch();
    while(ch != '"' || (*(lptr-1) == 92 && *(lptr-2) != 92)) {
      if(ch == NULL) {
        error("no quote");
        break;
        }
      keepch(gch());
      }
    gch();
    keepch('"');
    }
  else if(ch == 39) {
    keepch(39);
    gch();
    while(ch != 39 || (*(lptr-1) == 92 && *(lptr-2) != 92)) {
      if(ch == NULL) {
        error("no apostrophe");
        break;
        }
      keepch(gch());
      }
    gch();
    keepch(39);
    }
  else if(ch == '/' && nch == '*') {
```

```
      bump(2);
      while((ch == '*' && nch == '/') == 0) {
        if(ch) bump(1);
        else {
          ifline();
          if(eof) break;
          }
        }
      bump(2);
      }
    else if(an(ch)) {
      k = 0;
      while(an(ch) && k < NAMEMAX) {
        msname[k++] = ch;
        gch();
        }
      msname[k] = NULL;
      if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0)) {
        k = getint(cptr+NAMESIZE, 2);
        while(c = macq[k++]) keepch(c);
        while(an(ch)) gch();
        }
      else {
        k = 0;
        while(c = msname[k++]) keepch(c);
        }
      }
    else keepch(gch());
    }
  if(pptr >= LINEMAX) error("line too long");
  keepch(NULL);
  line = pline;
  bump(0);
  }

keepch(c)  char c; {
  if(pptr < LINEMAX) pline[++pptr] = c;
  }

ifline() {
  while(1) {
    inline();
    if(eof) return;
    if(match("#ifdef")) {
```

```
      ++iflevel;
      if(skiplevel) continue;
      symname(msname);
      if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0) == 0)
        skiplevel = iflevel;
      continue;
      }
    if(match("#ifndef")) {
      ++iflevel;
      if(skiplevel) continue;
      symname(msname);
      if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0))
        skiplevel = iflevel;
      continue;
      }
    if(match("#else")) {
      if(iflevel) {
        if(skiplevel == iflevel) skiplevel = 0;
        else if(skiplevel == 0)  skiplevel = iflevel;
        }
      else noiferr();
      continue;
      }
    if(match("#endif")) {
      if(iflevel) {
        if(skiplevel == iflevel) skiplevel = 0;
        --iflevel;
        }
      else noiferr();
      continue;
      }
    if(skiplevel) continue;
    if(ch == 0) continue;
    break;
    }
  }

inline() {            /* numerous revisions */
  int k, unit;
  poll(1);            /* allow operator interruption */
  if(input == EOF) openfile();
  if(eof) return;
  if((unit = input2) == EOF) unit = input;
  if(fgets(line, LINEMAX, unit) == NULL) {
```

```
    fclose(unit);
    if(input2 != EOF)
         input2 = EOF;
    else input  = EOF;
    *line = NULL;
    }
  else if(listfp) {
    if(listfp == output) fputc(';', output);
    fputs(line, listfp);
    }
  bump(0);
  }

inbyte()  {
  while(ch == 0) {
    if(eof) return 0;
    preprocess();
    }
  return gch();
  }

/********************* scanning functions ********************/

/*
** test if next input string is legal symbol name
*/
symname(sname) char *sname; {
  int k;char c;
  blanks();
  if(alpha(ch) == 0) return (*sname = 0);
  k = 0;
  while(an(ch)) {
    sname[k] = gch();
    if(k < NAMEMAX) ++k;
    }
  sname[k] = 0;
  return 1;
  }

need(str)  char *str; {
  if(match(str) == 0) error("missing token");
  }

ns()  {
```

```
  if(match(";") == 0) error("no semicolon");
  else errflag = 0;
  }

match(lit)  char *lit; {
  int k;
  blanks();
  if(k = streq(lptr, lit)) {
    bump(k);
    return 1;
    }
  return 0;
  }

streq(str1, str2)  char str1[], str2[]; {
  int k;
  k = 0;
  while (str2[k]) {
    if(str1[k] != str2[k]) return 0;
    ++k;
    }
  return k;
 }

amatch(lit, len)  char *lit; int len; {
  int k;
  blanks();
  if(k = astreq(lptr, lit, len)) {
    bump(k);
    return 1;
    }
  return 0;
 }

astreq(str1, str2, len)  char str1[], str2[]; int len; {
  int k;
  k = 0;
  while (k < len) {
    if(str1[k] != str2[k]) break;
    /*
    ** must detect end of symbol table names terminated by
    ** symbol length in binary
    */
    if(str2[k] < ' ') break;
```

```
    if(str1[k] < ' ') break;
    ++k;
    }
  if(an(str1[k]) || an(str2[k])) return 0;
  return k;
  }

nextop(list) char *list; {
  char op[4];
  opindex = 0;
  blanks();
  while(1) {
    opsize = 0;
    while(*list > ' ') op[opsize++] = *list++;
    op[opsize] = 0;
    if(opsize = streq(lptr, op))
      if(*(lptr+opsize) != '=' &&
         *(lptr+opsize) != *(lptr+opsize-1))
         return 1;
    if(*list) {
      ++list;
      ++opindex;
      }
    else return 0;
    }
  }

blanks() {
  while(1) {
    while(ch) {
      if(white()) gch();
      else return;
      }
    if(line == mline) return;
    preprocess();
    if(eof) break;
    }
  }

white() {
  avail(YES);  /* abort on stack/symbol table overflow */
  return (*lptr <= ' ' && *lptr);
  }
```

```
gch() {
  int c;
  if(c = ch) bump(1);
  return c;
  }

bump(n) int n; {
  if(n) lptr += n;
  else  lptr  = line;
  if(ch = nch = *lptr) nch = *(lptr+1);
  }

kill() {
  *line = 0;
  bump(0);
  }

skip() {
  if(an(inbyte()))
       while(an(ch)) gch();
  else while(an(ch) == 0) {
    if(ch == 0) break;
    gch();
    }
  blanks();
  }

endst() {
  blanks();
  return (streq(lptr, ";") || ch == 0);
  }

/*********** symbol table management functions ***********/

addsym(sname, id, type, size, value, lgpp, class)
  char *sname, id, type;  int size, value, *lgpp, class; {
  if(lgpp == &glbptr) {
    if(cptr2 = findglb(sname)) return cptr2;
    if(cptr == 0) {
      error("global symbol table overflow");
      return 0;
      }
    }
  else {
```

```
    if(locptr > (ENDLOC-SYMMAX)) {
      error("local symbol table overflow");
      abort(ERRCODE);
      }
    cptr = *lgpp;
    }
  cptr[IDENT] = id;
  cptr[TYPE]  = type;
  cptr[CLASS] = class;
  putint(size, cptr + SIZE, 2);
  putint(value, cptr + OFFSET, 2);
  cptr3 = cptr2 = cptr + NAME;
  while(an(*sname)) *cptr2++ = *sname++;
  if(lgpp == &locptr) {
    *cptr2 = cptr2 - cptr3;          /* set length */
    *lgpp = ++cptr2;
    }
  return cptr;
  }

/*
** search for symbol match
** on return cptr points to slot found or empty slot
*/
search(sname, buf, len, end, max, off)
  char *sname, *buf, *end;  int len, max, off; {
  cptr  =
  cptr2 = buf+((hash(sname)%(max-1))*len);
  while(*cptr != NULL) {
    if(astreq(sname, cptr+off, NAMEMAX)) return 1;
    if((cptr = cptr+len) >= end) cptr = buf;
    if(cptr == cptr2) return (cptr = 0);
    }
  return 0;
  }

hash(sname) char *sname; {
  int i, c;
  i = 0;
  while(c = *sname++) i = (i << 1) + c;
  return i;
  }

findglb(sname)  char *sname; {
```

```
  if(search(sname, STARTGLB, SYMMAX, ENDGLB, NUMGLBS, NAME))
    return cptr;
  return 0;
  }

findloc(sname)  char *sname;  {
  cptr = locptr - 1;  /* search backward for block locals */
  while(cptr > STARTLOC) {
    cptr = cptr - *cptr;
    if(astreq(sname, cptr, NAMEMAX)) return (cptr - NAME);
    cptr = cptr - NAME - 1;
    }
  return 0;
  }

nextsym(ent) char *ent; {
  ent = ent + NAME;
  while(*ent++ >= ' ');    /* find length byte */
  return ent;
  }

/******** while queue management functions *********/

addwhile(ptr)  int ptr[]; {
  int k;
  ptr[WQSP]   = csp;          /* and stk ptr */
  ptr[WQLOOP] = getlabel();  /* and looping label */
  ptr[WQEXIT] = getlabel();  /* and exit label */
  if(wqptr == WQMAX) {
    error("control statement nesting limit");
    abort(ERRCODE);
    }
  k = 0;
  while (k < WQSIZ) *wqptr++ = ptr[k++];
  }

readwhile(ptr) int *ptr; {
  if(ptr <= wq) {
    error("out of context");
    return 0;
    }
  else return (ptr - WQSIZ);
 }
```

```
delwhile() {
  if(wqptr > wq) wqptr -= WQSIZ;
  }

/****************** utility functions ********************/

/*
** test if c is alphabetic
*/
alpha(c)  char c; {
  return (isalpha(c) || c == '_');
  }

/*
** test if given character is alphanumeric
*/
an(c)  char c; {
  return (alpha(c) || isdigit(c));
  }


/*
** return next avail internal label number
*/
getlabel() {
  return(++nxtlab);
  }

/*
** get integer of length len from address addr
** (byte sequence set by "putint")
*/
getint(addr, len) char *addr; int len; {
  int i;
  i = *(addr + -len);  /* high order byte sign extended */
  while(len--) i = (i << 8) | *(addr + len) & 255;
  return i;
  }

/*
** put integer i of length len into address addr
** (low byte first)
*/
putint(i, addr, len) char *addr; int i, len; {
```

```
  while(len--) {
    *addr++ = i;
    i = i >> 8;
    }
  }

lout(line, fd) char *line; int fd; {
  fputs(line, fd);
  fputc(NEWLINE, fd);
  }

/****************** error functions ********************/

illname() {
  error("illegal symbol");
  skip();
  }

multidef(sname)  char *sname; {
  error("already defined");
  }

needlval() {
  error("must be lvalue");
  }

noiferr() {
  error("no matching #if...");
  errflag = 0;
  }

error(msg) char msg[]; {
  if(errflag) return;
  else errflag = 1;
  lout(line, stderr);
  errout(msg, stderr);
  if(alarm) fputc(7, stderr);
  if(pause) while(fgetc(stderr) != NEWLINE);
  if(listfp > 0) errout(msg, listfp);
  }

errout(msg, fp) char msg[]; int fp; {
  int k;
  k = line+2;
```

```
  while(k++ <= lptr) fputc(' ', fp);
  lout("/\\", fp);
  fputs("**** ", fp); lout(msg, fp);
  }
```

cc3.c

```
/*
** Small-C Compiler — Part 3 — Expression Analyzer.
** Copyright 1982, 1983, 1985, 1988 J. E. Hendrix
** All rights reserved.
*/

#include <stdio.h>
#include "cc.h"

#define ST 0   /* is[ST] - symbol table address, else 0 */
#define TI 1   /* is[TI] - type of indirect obj to fetch, else 0 */
#define TA 2   /* is[TA] - type of address, else 0 */
#define TC 3   /* is[TC] - type of constant (INT or UINT), else 0 */
#define CV 4   /* is[CV] - value of constant (+ auxiliary uses) */
#define OP 5   /* is[OP] - code of highest/last binary operator */
#define SA 6   /* is[SA] - stage address of "op 0" code, else 0 */

extern char
 *litq, *glbptr, *lptr,  ssname[NAMESIZE],  quote[2];
extern int
  ch,  csp,  litlab,  litptr,  nch,  op[16],  op2[16],
  opindex,  opsize, *snext;

/***************** lead-in functions ******************/

constexpr(val) int *val; {
  int const;
  int *before, *start;
  setstage(&before, &start);
  expression(&const, val);
  clearstage(before, 0);     /* scratch generated code */
  if(const == 0) error("must be constant expression");
  return const;
  }

expression(con, val) int *con, *val;  {
```

```
  int is[7];
  if(level1(is)) fetch(is);
  *con = is[TC];
  *val = is[CV];
  }

test(label, parens)  int label, parens;  {
  int is[7];
  int *before, *start;
  if(parens) need("(");
  while(1) {
    setstage(&before, &start);
    if(level1(is)) fetch(is);
    if(match(",")) clearstage(before, start);
    else break;
    }
  if(parens) need(")");
  if(is[TC]) {              /* constant expression */
    clearstage(before, 0);
    if(is[CV]) return;
    gen(JMPm, label);
    return;
    }
  if(is[SA]) {              /* stage address of "oper 0" code */
    switch(is[OP]) {        /* operator code */
      case EQ12:
      case LE12u: zerojump(EQ10f, label, is); break;
      case NE12:
      case GT12u: zerojump(NE10f, label, is); break;
      case GT12:  zerojump(GT10f, label, is); break;
      case GE12:  zerojump(GE10f, label, is); break;
      case GE12u: clearstage(is[SA], 0);      break;
      case LT12:  zerojump(LT10f, label, is); break;
      case LT12u: zerojump(JMPm,  label, is); break;
      case LE12:  zerojump(LE10f, label, is); break;
      default:    gen(NE10f, label);          break;
      }
    }
  else gen(NE10f, label);
  clearstage(before, start);
  }

/*
** test primary register against zero and jump if false
```

```
*/
zerojump(oper, label, is) int oper, label, is[]; {
  clearstage(is[SA], 0);       /* purge conventional code */
  gen(oper, label);
  }

/***************** precedence levels *****************/

level1(is) int is[];  {
  int k, is2[7], is3[2], oper, oper2;
  k = down1(level2, is);
  if(is[TC]) gen(GETw1n, is[CV]);
       if(match("|="))  {oper =        oper2 = OR12;}
  else if(match("^="))  {oper =        oper2 = XOR12;}
  else if(match("&="))  {oper =        oper2 = AND12;}
  else if(match("+="))  {oper =        oper2 = ADD12;}
  else if(match("-="))  {oper =        oper2 = SUB12;}
  else if(match("*="))  {oper = MUL12; oper2 = MUL12u;}
  else if(match("/="))  {oper = DIV12; oper2 = DIV12u;}
  else if(match("%="))  {oper = MOD12; oper2 = MOD12u;}
  else if(match(">>=")) {oper =        oper2 = ASR12;}
  else if(match("<<=")) {oper =        oper2 = ASL12;}
  else if(match("="))   {oper =        oper2 = 0;}
  else return k;
                        /* have an assignment operator */
  if(k == 0) {
    needlval();
    return 0;
    }
  is3[ST] = is[ST];
  is3[TI] = is[TI];
  if(is[TI]) {                          /* indirect target */
    if(oper) {                          /* ?= */
      gen(PUSH1, 0);                    /* save address */
      fetch(is);                        /* fetch left side */
      }
    down2(oper, oper2, level1, is, is2);  /* parse right side */
    if(oper) gen(POP2, 0);              /* retrieve address */
    }
  else {                                /* direct target */
    if(oper) {                          /* ?= */
      fetch(is);                        /* fetch left side */
      down2(oper, oper2, level1, is, is2); /* parse right side */
      }
```

```
    else {                                  /*  = */
      if(level1(is2)) fetch(is2);           /* parse right side */
      }
    }
  store(is3);                               /* store result */
  return 0;
  }

level2(is1)  int is1[]; {
  int is2[7], is3[7], k, flab, endlab, *before, *after;
  k = down1(level3, is1);                   /* expression 1 */
  if(match("?") == 0) return k;
  dropout(k, NE1Of, flab = getlabel(), is1);
  if(down1(level2, is2)) fetch(is2);        /* expression 2 */
  else if(is2[TC]) gen(GETw1n, is2[CV]);
  need(":");
  gen(JMPm, endlab = getlabel());
  gen(LABm, flab);
  if(down1(level2, is3)) fetch(is3);        /* expression 3 */
  else if(is3[TC]) gen(GETw1n, is3[CV]);
  gen(LABm, endlab);

  is1[TC] = is1[CV] = 0;
  if(is2[TC] && is3[TC]) {                  /* expr1 ? const2 : const3 */
    is1[TA] = is1[TI] = is1[SA] = 0;
    }
  else if(is3[TC]) {                        /* expr1 ? var2 : const3 */
    is1[TA] = is2[TA];
    is1[TI] = is2[TI];
    is1[SA] = is2[SA];
    }
  else if((is2[TC])                         /* expr1 ? const2 : var3 */
      || (is2[TA] == is3[TA])) {            /* expr1 ? same2 : same3 */
    is1[TA] = is3[TA];
    is1[TI] = is3[TI];
    is1[SA] = is3[SA];
    }
  else error("mismatched expressions");
  return 0;
  }

level3 (is) int is[]; {return skim("||", EQ1Of, 1, 0, level4,  is);}
level4 (is) int is[]; {return skim("&&", NE1Of, 0, 1, level5,  is);}
level5 (is) int is[]; {return down("|",         0, level6,  is);}
```

```
level6 (is) int is[]; {return down("^",          1, level7,  is);}
level7 (is) int is[]; {return down("&",          2, level8,  is);}
level8 (is) int is[]; {return down("== !=",      3, level9,  is);}
level9 (is) int is[]; {return down("<= >= < >",  5, level10, is);}
level10(is) int is[]; {return down(">> <<",      9, level11, is);}
level11(is) int is[]; {return down("+ -",       11, level12, is);}
level12(is) int is[]; {return down("* / %",     13, level13, is);}

level13(is)  int is[];  {
  int k;
  char *ptr;
  if(match("++")) {                 /* ++lval */
    if(level13(is) == 0) {
      needlval();
      return 0;
      }
    step(rINC1, is, 0);
    return 0;
    }
  else if(match("-")) {             /* -lval */
    if(level13(is) == 0) {
      needlval();
      return 0;
      }
    step(rDEC1, is, 0);
    return 0;
    }
  else if(match("~")) {             /* ~ */
    if(level13(is)) fetch(is);
    gen(COM1, 0);
    is[CV] =  ~is[CV];
    return (is[SA] = 0);
    }
  else if(match("!")) {             /* ! */
    if(level13(is)) fetch(is);
    gen(LNEG1, 0);
    is[CV] = ! is[CV];
    return (is[SA] = 0);
    }
  else if(match("-")) {             /* unary - */
    if(level13(is)) fetch(is);
    gen(ANEG1, 0);
    is[CV] = -is[CV];
    return (is[SA] = 0);
```

```
  }
else if(match("*")) {              /* unary * */
  if(level13(is)) fetch(is);
  if(ptr = is[ST]) is[TI] = ptr[TYPE];
  else             is[TI] = INT;
  is[SA] =       /* no (op 0) stage address */
  is[TA] =       /* not an address */
  is[TC] = 0;    /* not a constant */
  is[CV] = 1;    /* omit fetch() on func call */
  return 1;
  }
else if(amatch("sizeof", 6)) {     /* sizeof() */
  int sz, p;  char *ptr, sname[NAMESIZE];
  if(match("(")) p = 1;
  else           p = 0;
  sz = 0;
  if     (amatch("unsigned", 8))  sz = BPW;
  if     (amatch("int",      3))  sz = BPW;
  else if(amatch("char",     4))  sz = 1;
  if(sz) {if(match("*"))          sz = BPW;}
  else if(symname(sname)
       && ((ptr = findloc(sname)) ||
           (ptr = findglb(sname)))
       && ptr[IDENT] != FUNCTION
       && ptr[IDENT] != LABEL)    sz = getint(ptr+SIZE, 2);
  else if(sz == 0) error("must be object or type");
  if(p) need(")");
  is[TC] = INT;
  is[CV] = sz;
  is[TA] = is[TI] = is[ST] = 0;
  return 0;
  }
else if(match("&")) {              /* unary & */
  if(level13(is) == 0) {
    error("illegal address");
    return 0;
    }
  ptr = is[ST];
  is[TA] = ptr[TYPE];
  if(is[TI]) return 0;
  gen(POINT1m, ptr);
  is[TI] = ptr[TYPE];
  return 0;
  }
```

```
  else {
    k = level14(is);
    if(match("++")) {                /* lval++ */
      if(k == 0) {
        needlval();
        return 0;
        }
      step(rINC1, is, rDEC1);
      return 0;
      }
    else if(match("-")) {           /* lval- */
      if(k == 0) {
        needlval();
        return 0;
        }
      step(rDEC1, is, rINC1);
      return 0;
      }
    else return k;
    }
  }

level14(is)  int *is; {
  int k, const, val;
  char *ptr, *before, *start;
  k = primary(is);
  ptr = is[ST];
  blanks();
  if(ch == '[' || ch == '(') {
    int is2[7];                      /* allocate only if needed */
    while(1) {
      if(match("[")) {               /* [subscript] */
        if(ptr == 0) {
          error("can't subscript");
          skip();
          need("]");
          return 0;
          }
        if(is[TA]) {if(k) fetch(is);}
        else       {error("can't subscript"); k = 0;}
        setstage(&before, &start);
        is2[TC] = 0;
        down2(0, 0, level1, is2, is2);
        need("]");
```

```
        if(is2[TC]) {
          clearstage(before, 0);
          if(is2[CV]) {             /* only add if non-zero */
            if(ptr[TYPE] >> 2 == BPW)
                 gen(GETw2n, is2[CV] << LBPW);
            else gen(GETw2n, is2[CV]);
            gen(ADD12, 0);
            }
          }
        else {
          if(ptr[TYPE] >> 2 == BPW) gen(DBL1, 0);
          gen(ADD12, 0);
          }
        is[TA] = 0;
        is[TI] = ptr[TYPE];
        k = 1;
        }
      else if(match("(")) {         /* function(...) */
        if(ptr == 0) callfunc(0);
        else if(ptr[IDENT] != FUNCTION) {
          if(k && !is[CV]) fetch(is);
          callfunc(0);
          }
        else callfunc(ptr);
        k = is[ST] = is[TC] = is[CV] = 0;
        }
      else return k;
      }
    }
  if(ptr && ptr[IDENT] == FUNCTION) {
    gen(POINT1m, ptr);
    is[ST] = 0;
    return 0;
    }
  return k;
  }

primary(is)  int *is; {
  char *ptr, sname[NAMESIZE];
  int k;
  if(match("(")) {                  /* (subexpression) */
    do k = level1(is); while(match(","));
    need(")");
    return k;
```

```
    }
  putint(0, is, 7 << LBPW);          /* clear "is" array */
  if(symname(sname)) {               /* is legal symbol */
    if(ptr = findloc(sname)) {       /* is local */
      if(ptr[IDENT] == LABEL) {
        experr();
        return 0;
        }
      gen(POINT1s, getint(ptr+OFFSET, 2));
      is[ST] = ptr;
      is[TI] = ptr[TYPE];
      if(ptr[IDENT] == ARRAY) {
        is[TA] = ptr[TYPE];
        return 0;
        }
      if(ptr[IDENT] == POINTER) {
        is[TI] = UINT;
        is[TA] = ptr[TYPE];
        }
      return 1;
      }
    if(ptr = findglb(sname)) {       /* is global */
      is[ST] = ptr;
      if(ptr[IDENT] != FUNCTION) {
        if(ptr[IDENT] == ARRAY) {
          gen(POINT1m, ptr);
          is[TI] =
          is[TA] = ptr[TYPE];
          return 0;
          }
        if(ptr[IDENT] == POINTER)
          is[TA] = ptr[TYPE];
        return 1;
        }
      }
    else is[ST] = addsym(sname, FUNCTION, INT, 0, 0, &glbptr, AUTOEXT);
    return 0;
    }
  if(constant(is) == 0) experr();
  return 0;
  }

experr() {
  error("invalid expression");
```

```
  gen(GETw1n, 0);
  skip();
  }

callfunc(ptr)  char *ptr; {       /* symbol table entry or 0 */
  int nargs, const, val;
  nargs = 0;
  blanks();                       /* already saw open paren */
  while(streq(lptr, ")") == 0) {
    if(endst()) break;
    if(ptr) {
      expression(&const, &val);
      gen(PUSH1, 0);
      }
    else {
      gen(PUSH1, 0);
      expression(&const, &val);
      gen(SWAP1s, 0);             /* don't push addr */
      }
    nargs = nargs + BPW;          /* count args*BPW */
    if(match(",") == 0) break;
    }
  need(")");
  if(streq(ptr + NAME, "CCARGC") == 0) gen(ARGCNTn, nargs >> LBPW);
  if(ptr) gen(CALLm, ptr);
  else    gen(CALL1, 0);
  gen(ADDSP, csp + nargs);
  }

/*
** true if is2's operand should be doubled
*/
double(oper, is1, is2) int oper, is1[], is2[]; {
  if((oper != ADD12 && oper != SUB12)
  || (is1[TA] >> 2 != BPW)
  || (is2[TA])) return 0;
  return 1;
  }

step(oper, is, oper2) int oper, is[], oper2; {
  fetch(is);
  gen(oper, is[TA] ? (is[TA] >> 2) : 1);
  store(is);
  if(oper2) gen(oper2, is[TA] ? (is[TA] >> 2) : 1);
```

```
    }

store(is)  int is[]; {
    char *ptr;
    if(is[TI]) {                        /* putstk */
      if(is[TI] >> 2 == 1)
           gen(PUTbp1, 0);
      else gen(PUTwp1, 0);
      }
    else {                              /* putmem */
      ptr = is[ST];
      if(ptr[IDENT] != POINTER
      && ptr[TYPE] >> 2 == 1)
           gen(PUTbm1, ptr);
      else gen(PUTwm1, ptr);
      }
    }

fetch(is) int is[]; {
    char *ptr;
    ptr = is[ST];
    if(is[TI]) {                                        /* indirect */
      if(is[TI] >> 2 == BPW)     gen(GETw1p,  0);
      else {
        if(ptr[TYPE] & UNSIGNED) gen(GETb1pu, 0);
        else                     gen(GETb1p,  0);
        }
      }
    else {                                             /* direct */
      if(ptr[IDENT] == POINTER
      || ptr[TYPE] >> 2 == BPW)  gen(GETw1m,  ptr);
      else {
        if(ptr[TYPE] & UNSIGNED) gen(GETb1mu, ptr);
        else                     gen(GETb1m,  ptr);
        }
      }
    }

constant(is)  int is[]; {
    int offset;
    if     (is[TC] = number(is + CV)) gen(GETw1n,  is[CV]);
    else if(is[TC] = chrcon(is + CV)) gen(GETw1n,  is[CV]);
    else if(string(&offset))          gen(POINT1l, offset);
    else return 0;
```

```
  return 1;
  }

number(value)  int *value; {
  int k, minus;
  k = minus = 0;
  while(1) {
    if(match("+")) ;
    else if(match("-")) minus = 1;
    else break;
    }
  if(isdigit(ch) == 0) return 0;
  if(ch == '0') {
    while(ch == '0') inbyte();
    if(toupper(ch) == 'X') {
      inbyte();
      while(isxdigit(ch)) {
        if(isdigit(ch))
             k = k*16 + (inbyte() - '0');
        else k = k*16 + 10 + (toupper(inbyte()) - 'A');
        }
      }
    else while (ch >= '0' && ch <= '7')
      k = k*8 + (inbyte() - '0');
    }
  else while (isdigit(ch)) k = k*10 + (inbyte() - '0');
  if(minus) {
    *value = -k;
    return (INT);
    }
  if((*value = k) < 0) return (UINT);
  else                 return (INT);
  }

chrcon(value)  int *value; {
  int k;
  k = 0;
  if(match("'") == 0) return 0;
  while(ch != '\'') k = (k << 8) + (litchar() & 255);
  gch();
  *value = k;
  return (INT);
  }
```

```
string(offset) int *offset; {
  char c;
  if(match(quote) == 0) return 0;
  *offset = litptr;
  while (ch != '"') {
    if(ch == 0) break;
    stowlit(litchar(), 1);
    }
  gch();
  litq[litptr++] = 0;
  return 1;
  }

stowlit(value, size) int value, size; {
  if((litptr+size) >= LITMAX) {
    error("literal queue overflow");
    abort(ERRCODE);
    }
  putint(value, litq+litptr, size);
  litptr += size;
  }

litchar() {
  int i, oct;
  if(ch != '\\'  || nch == 0) return gch();
  gch();
  switch(ch) {
    case 'n': gch(); return NEWLINE;
    case 't': gch(); return  9;  /* HT */
    case 'b': gch(); return  8;  /* BS */
    case 'f': gch(); return 12;  /* FF */
    }
  i = 3;
  oct = 0;
  while((i--) > 0 && ch >= '0' && ch <= '7')
    oct = (oct << 3) + gch() - '0';
  if(i == 2) return gch();
  else       return oct;
  }

/***************** pipeline functions *****************/

/*
** skim over terms adjoining || and && operators
```

```
*/
skim(opstr, tcode, dropval, endval, level, is)
  char *opstr;
  int tcode, dropval, endval, (*level)(), is[]; {
  int k, droplab, endlab;
  droplab = 0;
  while(1) {
    k = down1(level, is);
    if(nextop(opstr)) {
      bump(opsize);
      if(droplab == 0) droplab = getlabel();
      dropout(k, tcode, droplab, is);
      }
    else if(droplab) {
      dropout(k, tcode, droplab, is);
      gen(GETw1n, endval);
      gen(JMPm, endlab = getlabel());
      gen(LABm, droplab);
      gen(GETw1n, dropval);
      gen(LABm, endlab);
      is[TI] = is[TA] = is[TC] = is[CV] = is[SA] = 0;
      return 0;
      }
    else return k;
    }
  }

/*
** test for early dropout from || or && sequences
*/
dropout(k, tcode, exit1, is)
  int k, tcode, exit1, is[]; {
  if(k) fetch(is);
  else if(is[TC]) gen(GETw1n, is[CV]);
  gen(tcode, exit1);          /* jumps on false */
  }

/*
** drop to a lower level
*/
down(opstr, opoff, level, is)
  char *opstr;  int opoff, (*level)(), is[]; {
  int k;
  k = down1(level, is);
```

```
  if(nextop(opstr) == 0) return k;
  if(k) fetch(is);
  while(1) {
    if(nextop(opstr)) {
      int is2[7];     /* allocate only if needed */
      bump(opsize);
      opindex += opoff;
      down2(op[opindex], op2[opindex], level, is, is2);
      }
    else return 0;
    }
  }

/*
** unary drop to a lower level
*/
down1(level, is) int (*level)(), is[]; {
  int k, *before, *start;
  setstage(&before, &start);
  k = (*level)(is);
  if(is[TC]) clearstage(before, 0);  /* load constant later */
  return k;
  }

/*
** binary drop to a lower level
*/
down2(oper, oper2, level, is, is2)
  int oper, oper2, (*level)(), is[], is2[]; {
  int *before, *start;
  char *ptr;
  setstage(&before, &start);
  is[SA] = 0;                     /* not "... op 0" syntax */
  if(is[TC]) {                    /* consant op unknown */
    if(down1(level, is2)) fetch(is2);
    if(is[CV] == 0) is[SA] = snext;
    gen(GETw2n, is[CV] << double(oper, is2, is));
    }
  else {                          /* variable op unknown */
    gen(PUSH1, 0);                /* at start in the buffer */
    if(down1(level, is2)) fetch(is2);
    if(is2[TC]) {                 /* variable op constant */
      if(is2[CV] == 0) is[SA] = start;
      csp += BPW;                 /* adjust stack and */
```

```
    clearstage(before, 0);      /* discard the PUSH */
    if(oper == ADD12) {         /* commutative */
      gen(GETw2n, is2[CV] << double(oper, is, is2));
      }
    else {                      /* non-commutative */
      gen(MOVE21, 0);
      gen(GETw1n, is2[CV] << double(oper, is, is2));
      }
    }
  else {                        /* variable op variable */
    gen(POP2, 0);
    if(double(oper, is, is2)) gen(DBL1, 0);
    if(double(oper, is2, is)) gen(DBL2, 0);
    }
  }
if(oper) {
  if(nosign(is) || nosign(is2)) oper = oper2;
  if(is[TC] = is[TC] & is2[TC]) {               /* constant result */
    is[CV] = calc(is[CV], oper, is2[CV]);
    clearstage(before, 0);
    if(is2[TC] == UINT) is[TC] = UINT;
    }
  else {                                        /* variable result */
    gen(oper, 0);
    if(oper == SUB12
    && is [TA] >> 2 == BPW
    && is2[TA] >> 2 == BPW) { /* difference of two word addresses */
      gen(SWAP12, 0);
      gen(GETw1n, 1);
      gen(ASR12, 0);          /* div by 2 */
      }
    is[OP] = oper;            /* identify the operator */
    }
  if(oper == SUB12 || oper == ADD12) {
    if(is[TA] && is2[TA])     /*  addr +/- addr */
      is[TA] = 0;
    else if(is2[TA]) {        /* value +/- addr */
      is[ST] = is2[ST];
      is[TI] = is2[TI];
      is[TA] = is2[TA];
      }
    }
  if(is[ST] == 0 || ((ptr = is2[ST]) && (ptr[TYPE] & UNSIGNED)))
    is[ST] = is2[ST];
```

```
    }
  }

/*
** unsigned operand?
*/
nosign(is) int is[]; {
  char *ptr;
  if(is[TA]
  || is[TC] == UINT
  || ((ptr = is[ST]) && (ptr[TYPE] & UNSIGNED))
    ) return 1;
  return 0;
  }

/*
** calculate signed constant result
*/
calc(left, oper, right) int left, oper, right; {
  switch(oper) {
    case ADD12: return (left  +  right);
    case SUB12: return (left  -  right);
    case MUL12: return (left  *  right);
    case DIV12: return (left  /  right);
    case MOD12: return (left  %  right);
    case EQ12:  return (left  == right);
    case NE12:  return (left  != right);
    case LE12:  return (left  <= right);
    case GE12:  return (left  >= right);
    case LT12:  return (left  <  right);
    case GT12:  return (left  >  right);
    case AND12: return (left  &  right);
    case OR12:  return (left  |  right);
    case XOR12: return (left  ^  right);
    case ASR12: return (left  >> right);
    case ASL12: return (left  << right);
    }
  return (calc2(left, oper, right));
  }

/*
** calcualte unsigned constant result
*/
calc2(left, oper, right) unsigned left, right; int oper; {
```

```
  switch(oper) {
    case MUL12u: return (left  *  right);
    case DIV12u: return (left  /  right);
    case MOD12u: return (left  %  right);
    case LE12u:  return (left  <= right);
    case GE12u:  return (left  >= right);
    case LT12u:  return (left  <  right);
    case GT12u:  return (left  >  right);
    }
  return (0);
  }
```

cc4.c

```
/*
** Small-C Compiler — Part 4 — Back End.
** Copyright 1982, 1983, 1985, 1988 J. E. Hendrix
** All rights reserved.
*/
#include <stdio.h>
#include "cc.h"

/* #define DISOPT */       /* display optimizations values */

/************************** externals ****************************/

extern char
  *cptr, *macn, *litq, *symtab, optimize, ssname[NAMESIZE];

extern int
  *stage, litlab, litptr, csp, output, oldseg, usexpr,
  *snext, *stail, *slast;


/***************** optimizer command definitions *****************/

            /*     —      p-codes must not overlap these */
#define any     0x00FF   /* matches any p-code */
#define _pop    0x00FE   /* matches if corresponding POP2 exists */
#define pfree   0x00FD   /* matches if pri register free */
#define sfree   0x00FC   /* matches if sec register free */
#define comm    0x00FB   /* matches if registers are commutative */
```

```
            /*     -      these digits are reserved for n */
#define go      0x0100   /* go n entries */
#define gc      0x0200   /* get code from n entries away */
#define gv      0x0300   /* get value from n entries away */
#define sum     0x0400   /* add value from nth entry away */
#define neg     0x0500   /* negate the value */
#define ife     0x0600   /* if value == n do commands to next 0 */
#define ifl     0x0700   /* if value <  n do commands to next 0 */
#define swv     0x0800   /* swap value with value n entries away */
#define topop   0x0900   /* moves |code and current value to POP2 */

#define p1      0x0001   /* plus 1 */
#define p2      0x0002   /* plus 2 */
#define p3      0x0003   /* plus 3 */
#define p4      0x0004   /* plus 4 */
#define m1      0x00FF   /* minus 1 */
#define m2      0x00FE   /* minus 2 */
#define m3      0x00FD   /* minus 3 */
#define m4      0x00FC   /* minus 4 */

#define PRI      0030    /* primary register bits */
#define SEC      0003    /* secondary register bits */
#define USES     0011    /* use register contents */
#define ZAPS     0022    /* zap register contents */
#define PUSHES   0100    /* pushes onto the stack */
#define COMMUTES 0200    /* commutative p-code */

/******************* optimizer command lists *********************/

int
  seq00[] = {0,ADD12,MOVE21,0,                          /* ADD21 */
             go|p1,ADD21,0},

  seq01[] = {0,ADD1n,0,                                 /* rINC1 or rDEC1 ?
*/
             ifl|m2,0,ifl|0,rDEC1,neg,0,ifl|p3,rINC1,0,0},

  seq02[] = {0,ADD2n,0,                                 /* rINC2 or rDEC2 ?
*/
             ifl|m2,0,ifl|0,rDEC2,neg,0,ifl|p3,rINC2,0,0},

  seq03[] = {0,rDEC1,PUTbp1,rINC1,0,                    /* SUBbpn or DECbp */
             go|p2,ife|p1,DECbp,0,SUBbpn,0},
```

```
seq04[] = {0,rDEC1,PUTwp1,rINC1,0,                    /* SUBwpn or DECwp */
           go|p2,ife|p1,DECwp,0,SUBwpn,0},

seq05[] = {0,rDEC1,PUTbm1,rINC1,0,                    /* SUB_m_ COMMAn */
           go|p1,SUB_m_,go|p1,COMMAn,go|m1,0},

seq06[] = {0,rDEC1,PUTwm1,rINC1,0,                    /* SUB_m_ COMMAn */
           go|p1,SUB_m_,go|p1,COMMAn,go|m1,0},

seq07[] = {0,GETw1m,GETw2n,ADD12,MOVE21,GETb1p,0,     /* GETw2m GETb1p */
           go|p4,gv|m3,go|m1,GETw2m,gv|m3,0},

seq08[] = {0,GETw1m,GETw2n,ADD12,MOVE21,GETb1pu,0,    /* GETw2m GETb1pu */
           go|p4,gv|m3,go|m1,GETw2m,gv|m3,0},

seq09[] = {0,GETw1m,GETw2n,ADD12,MOVE21,GETw1p,0,     /* GETw2m GETw1p */
           go|p4,gv|m3,go|m1,GETw2m,gv|m3,0},

seq10[] = {0,GETw1m,GETw2m,SWAP12,0,                  /* GETw2m GETw1m */
           go|p2,GETw1m,gv|m1,go|m1,gv|m1,0},

seq11[] = {0,GETw1m,MOVE21,0,                         /* GETw2m */
           go|p1,GETw2m,gv|m1,0},

seq12[] = {0,GETw1m,PUSH1,pfree,0,                    /* PUSHm */
           go|p1,PUSHm,gv|m1,0},

seq13[] = {0,GETw1n,PUTbm1,pfree,0,                   /* PUT_m_ COMMAn */
           PUT_m_,go|p1,COMMAn,go|m1,swv|p1,0},

seq14[] = {0,GETw1n,PUTwm1,pfree,0,                   /* PUT_m_ COMMAn */
           PUT_m_,go|p1,COMMAn,go|m1,swv|p1,0},

seq15[] = {0,GETw1p,PUSH1,pfree,0,                    /* PUSHp */
           go|p1,PUSHp,gv|m1,0},

seq16[] = {0,GETw1s,GETw2n,ADD12,MOVE21,0,            /* GETw2s ADD2n */
           go|p3,ADD2n,gv|m2,go|m1,GETw2s,gv|m2,0},

seq17[] = {0,GETw1s,GETw2s,SWAP12,0,                  /* GETw2s GETw1s */
           go|p2,GETw1s,gv|m1,go|m1,GETw2s,gv|m1,0},

seq18[] = {0,GETw1s,MOVE21,0,                         /* GETw2s */
           go|p1,GETw2s,gv|m1,0},
```

```
seq19[] = {0,GETw2m,GETw1n,SWAP12,SUB12,0,          /* GETw1m SUB1n */
           go|p3,SUB1n,gv|m2,go|m1,GETw1m,gv|m2,0},

seq20[] = {0,GETw2n,ADD12,0,                        /* ADD1n */
           go|p1,ADD1n,gv|m1,0},

seq21[] = {0,GETw2s,GETw1n,SWAP12,SUB12,0,          /* GETw1s SUB1n */
           go|p3,SUB1n,gv|m2,go|m1,GETw1s,gv|m2,0},

seq22[] = {0,rINC1,PUTbm1,rDEC1,0,                  /* ADDm_ COMMAn */
           go|p1,ADDm_,go|p1,COMMAn,go|m1,0},

seq23[] = {0,rINC1,PUTwm1,rDEC1,0,                  /* ADDm_ COMMAn */
           go|p1,ADDm_,go|p1,COMMAn,go|m1,0},

seq24[] = {0,rINC1,PUTbp1,rDEC1,0,                  /* ADDbpn or INCbp */
           go|p2,ife|p1,INCbp,0,ADDbpn,0},

seq25[] = {0,rINC1,PUTwp1,rDEC1,0,                  /* ADDwpn or INCwp */
           go|p2,ife|p1,INCwp,0,ADDwpn,0},

seq26[] = {0,MOVE21,GETw1n,SWAP12,SUB12,0,          /* SUB1n */
           go|p3,SUB1n,gv|m2,0},

seq27[] = {0,MOVE21,GETw1n,comm,0,                  /* GETw2n comm */
           go|p1,GETw2n,0},

seq28[] = {0,POINT1m,GETw2n,ADD12,MOVE21,0,         /* POINT2m_ PLUSn */
           go|p3,PLUSn,gv|m2,go|m1,POINT2m_,gv|m2,0},

seq29[] = {0,POINT1m,MOVE21,pfree,0,                /* POINT2m */
           go|p1,POINT2m,gv|m1,0},

seq30[] = {0,POINT1m,PUSH1,pfree,_pop,0,            /* ... POINT2m */
           topop|POINT2m,go|p2,0},

seq31[] = {0,POINT1s,GETw2n,ADD12,MOVE21,GETb1p,0, /* GETb1s */
           sum|p1,go|p4,GETb1s,gv|m4,0},


seq32[] = {0,POINT1s,GETw2n,ADD12,MOVE21,GETb1pu,0,/* GETb1su */
           sum|p1,go|p4,GETb1su,gv|m4,0},
```

```
seq33[] = {0,POINT1s,GETw2n,ADD12,MOVE21,GETw1p,0, /* GETw1s */
           sum|p1,go|p4,GETw1s,gv|m4,0},

seq34[] = {0,POINT1s,PUSH1,MOVE21,0,               /* POINT2s PUSH2 */
           go|p1,POINT2s,gv|m1,go|p1,PUSH2,go|m1,0},

seq35[] = {0,POINT1s,PUSH1,pfree,_pop,0,           /* ... POINT2s */
           topop|POINT2s,go|p2,0},

seq36[] = {0,POINT1s,MOVE21,0,                     /* POINT2s */
           go|p1,POINT2s,gv|m1,0},

seq37[] = {0,POINT2m,GETb1p,sfree,0,               /* GETb1m */
           go|p1,GETb1m,gv|m1,0},

seq38[] = {0,POINT2m,GETb1pu,sfree,0,              /* GETb1mu */
           go|p1,GETb1mu,gv|m1,0},

seq39[] = {0,POINT2m,GETw1p,sfree,0,               /* GETw1m */
           go|p1,GETw1m,gv|m1,0},

seq40[] = {0,POINT2m_,PLUSn,GETw1p,sfree,0,        /* GETw1m_ PLUSn */
           go|p2,gc|m1,gv|m1,go|m1,GETw1m_,gv|m1,0},

seq41[] = {0,POINT2s,GETb1p,sfree,0,               /* GETb1s */
           sum|p1,go|p1,GETb1s,gv|m1,0},

seq42[] = {0,POINT2s,GETb1pu,sfree,0,              /* GETb1su */
           sum|p1,go|p1,GETb1su,gv|m1,0},

seq43[] = {0,POINT2s,GETw1p,PUSH1,pfree,0,         /* PUSHs */
           sum|p1,go|p2,PUSHs,gv|m2,0},

seq44[] = {0,POINT2s,GETw1p,sfree,0,               /* GETw1s */
           sum|p1,go|p1,GETw1s,gv|m1,0},

seq45[] = {0,PUSH1,any,POP2,0,                     /* MOVE21 any */
           go|p2,gc|m1,gv|m1,go|m1,MOVE21,0},

seq46[] = {0,PUSHm,_pop,0,                         /* ... GETw2m */
           topop|GETw2m,go|p1,0},

seq47[] = {0,PUSHp,any,POP2,0,                     /* GETw2p ... */
           go|p2,gc|m1,gv|m1,go|m1,GETw2p,gv|m1,0},
```

```
  seq48[] = {0,PUSHs,_pop,0,                          /* ... GETw2s */
             topop|GETw2s,go|p1,0},

  seq49[] = {0,SUB1n,0,                               /* rDEC1 or rINC1 ?
*/
             if1|m2,0,if1|0,rINC1,neg,0,if1|p3,rDEC1,0,0};

#define HIGH_SEQ  49
int seq[HIGH_SEQ + 1];
setseq() {
  seq[ 0] = seq00;  seq[ 1] = seq01;  seq[ 2] = seq02;  seq[ 3] = seq03;
  seq[ 4] = seq04;  seq[ 5] = seq05;  seq[ 6] = seq06;  seq[ 7] = seq07;
  seq[ 8] = seq08;  seq[ 9] = seq09;  seq[10] = seq10;  seq[11] = seq11;
  seq[12] = seq12;  seq[13] = seq13;  seq[14] = seq14;  seq[15] = seq15;
  seq[16] = seq16;  seq[17] = seq17;  seq[18] = seq18;  seq[19] = seq19;
  seq[20] = seq20;  seq[21] = seq21;  seq[22] = seq22;  seq[23] = seq23;
  seq[24] = seq24;  seq[25] = seq25;  seq[26] = seq26;  seq[27] = seq27;
  seq[28] = seq28;  seq[29] = seq29;  seq[30] = seq30;  seq[31] = seq31;
  seq[32] = seq32;  seq[33] = seq33;  seq[34] = seq34;  seq[35] = seq35;
  seq[36] = seq36;  seq[37] = seq37;  seq[38] = seq38;  seq[39] = seq39;
  seq[40] = seq40;  seq[41] = seq41;  seq[42] = seq42;  seq[43] = seq43;
  seq[44] = seq44;  seq[45] = seq45;  seq[46] = seq46;  seq[47] = seq47;
  seq[48] = seq48;  seq[49] = seq49;
  }

/***************** assembly-code strings ******************/

int code[PCODES];

/*
** First byte contains flag bits indicating:
**    the value in ax is needed (010) or zapped (020)
**    the value in bx is needed (001) or zapped (002)
*/
setcodes() {
  setseq();
  code[ADD12]   = "\211ADD AX,BX\n";
  code[ADD1n]   = "\010?ADD AX,<n>\n??";
  code[ADD21]   = "\211ADD BX,AX\n";
  code[ADD2n]   = "\010?ADD BX,<n>\n??";
  code[ADDbpn]  = "\001ADD BYTE PTR [BX],<n>\n";
  code[ADDwpn]  = "\001ADD WORD PTR [BX],<n>\n";
```

```
code[ADDm_]   = "\000ADD <m>";
code[ADDSP]   = "\000?ADD SP,<n>\n??";
code[AND12]   = "\211AND AX,BX\n";
code[ANEG1]   = "\010NEG AX\n";
code[ARGCNTn] = "\000?MOV CL,<n>?XOR CL,CL?\n";
code[ASL12]   = "\011MOV CX,AX\nMOV AX,BX\nSAL AX,CL\n";
code[ASR12]   = "\011MOV CX,AX\nMOV AX,BX\nSAR AX,CL\n";
code[CALL1]   = "\010CALL AX\n";
code[CALLm]   = "\020CALL <m>\n";
code[BYTE_]   = "\000 DB ";
code[BYTEn]   = "\000 DB <n>\n";
code[BYTEr0]  = "\000 DB <n> DUP(0)\n";
code[COM1]    = "\010NOT AX\n";
code[COMMAn]  = "\000,<n>\n";
code[DBL1]    = "\010SHL AX,1\n";
code[DBL2]    = "\001SHL BX,1\n";
code[DECbp]   = "\001DEC BYTE PTR [BX]\n";
code[DECwp]   = "\001DEC WORD PTR [BX]\n";
code[DIV12]   = "\011CWD\nIDIV BX\n";                 /* see gen() */
code[DIV12u]  = "\011XOR DX,DX\nDIV BX\n";            /* see gen() */
code[ENTER]   = "\100PUSH BP\nMOV BP,SP\n";
code[EQ10f]   = "\010OR AX,AX\nJE $+5\nJMP _<n>\n";
code[EQ12]    = "\211CALL __EQ\n";
code[GE10f]   = "\010OR AX,AX\nJGE $+5\nJMP _<n>\n";
code[GE12]    = "\011CALL __GE\n";
code[GE12u]   = "\011CALL __UGE\n";
code[GETb1m]  = "\020MOV AL,<m>\nCBW\n";
code[GETb1mu] = "\020MOV AL,<m>\nXOR AH,AH\n";
code[GETb1p]  = "\021MOV AL,?<n>??[BX]\nCBW\n";       /* see gen() */
code[GETb1pu] = "\021MOV AL,?<n>??[BX]\nXOR AH,AH\n"; /* see gen() */
code[GETb1s]  = "\020MOV AL,<n>[BP]\nCBW\n";
code[GETb1su] = "\020MOV AL,<n>[BP]\nXOR AH,AH\n";
code[GETw1m]  = "\020MOV AX,<m>\n";
code[GETw1m_] = "\020MOV AX,<m>";
code[GETw1n]  = "\020?MOV AX,<n>?XOR AX,AX?\n";
code[GETw1p]  = "\021MOV AX,?<n>??[BX]\n";            /* see gen() */
code[GETw1s]  = "\020MOV AX,<n>[BP]\n";
code[GETw2m]  = "\002MOV BX,<m>\n";
code[GETw2n]  = "\002?MOV BX,<n>?XOR BX,BX?\n";
code[GETw2p]  = "\021MOV BX,?<n>??[BX]\n";
code[GETw2s]  = "\002MOV BX,<n>[BP]\n";
code[GT10f]   = "\010OR AX,AX\nJG $+5\nJMP _<n>\n";
code[GT12]    = "\010CALL __GT\n";
code[GT12u]   = "\011CALL __UGT\n";
```

```
code[INCbp]   = "\001INC BYTE PTR [BX]\n";
code[INCwp]   = "\001INC WORD PTR [BX]\n";
code[WORD_]   = "\000 DW ";
code[WORDn]   = "\000 DW <n>\n";
code[WORDr0]  = "\000 DW <n> DUP(0)\n";
code[JMPm]    = "\000JMP _<n>\n";
code[LABm]    = "\000_<n>:\n";
code[LE10f]   = "\0100R AX,AX\nJLE $+5\nJMP _<n>\n";
code[LE12]    = "\011CALL __LE\n";
code[LE12u]   = "\011CALL __ULE\n";
code[LNEG1]   = "\010CALL __LNEG\n";
code[LT10f]   = "\0100R AX,AX\nJL $+5\nJMP _<n>\n";
code[LT12]    = "\011CALL __LT\n";
code[LT12u]   = "\011CALL __ULT\n";
code[MOD12]   = "\011CWD\nIDIV BX\nMOV AX,DX\n";      /* see gen() */
code[MOD12u]  = "\011XOR DX,DX\nDIV BX\nMOV AX,DX\n"; /* see gen() */
code[MOVE21]  = "\012MOV BX,AX\n";
code[MUL12]   = "\211IMUL BX\n";
code[MUL12u]  = "\211MUL BX\n";
code[NE10f]   = "\0100R AX,AX\nJNE $+5\nJMP _<n>\n";
code[NE12]    = "\211CALL __NE\n";
code[NEARm]   = "\000 DW _<n>\n";
code[OR12]    = "\211OR AX,BX\n";
code[PLUSn]   = "\000?+<n>??\n";
code[POINT1l] = "\020MOV AX,OFFSET _<l>+<n>\n";
code[POINT1m] = "\020MOV AX,OFFSET <m>\n";
code[POINT1s] = "\020LEA AX,<n>[BP]\n";
code[POINT2m] = "\002MOV BX,OFFSET <m>\n";
code[POINT2m_]= "\002MOV BX,OFFSET <m>";
code[POINT2s] = "\002LEA BX,<n>[BP]\n";
code[POP2]    = "\002POP BX\n";
code[PUSH1]   = "\110PUSH AX\n";
code[PUSH2]   = "\101PUSH BX\n";
code[PUSHm]   = "\100PUSH <m>\n";
code[PUSHp]   = "\100PUSH ?<n>??[BX]\n";
code[PUSHs]   = "\100PUSH ?<n>??[BP]\n";
code[PUT_m_]  = "\000MOV <m>";
code[PUTbm1]  = "\010MOV <m>,AL\n";
code[PUTbp1]  = "\011MOV [BX],AL\n";
code[PUTwm1]  = "\010MOV <m>,AX\n";
code[PUTwp1]  = "\011MOV [BX],AX\n";
code[rDEC1]   = "\010#DEC AX\n#";
code[rDEC2]   = "\010#DEC BX\n#";
code[REFm]    = "\000_<n>";
```

```
  code[RETURN]  = "\000?MOV SP,BP\n??POP BP\nRET\n";
  code[rINC1]   = "\010#INC AX\n#";
  code[rINC2]   = "\010#INC BX\n#";
  code[SUB_m_]  = "\000SUB <m>";
  code[SUB12]   = "\011SUB AX,BX\n";                    /* see gen() */
  code[SUB1n]   = "\010?SUB AX,<n>\n??";
  code[SUBbpn]  = "\001SUB BYTE PTR [BX],<n>\n";
  code[SUBwpn]  = "\001SUB WORD PTR [BX],<n>\n";
  code[SWAP12]  = "\011XCHG AX,BX\n";
  code[SWAP1s]  = "\012POP BX\nXCHG AX,BX\nPUSH BX\n";
  code[SWITCH]  = "\012CALL __SWITCH\n";
  code[XOR12]   = "\211XOR AX,BX\n";
  }

/***************** code generation functions *****************/

/*
** print all assembler info before any code is generated
** and ensure that the segments appear in the correct order.
*/
header()  {
  toseg(CODESEG);
  outline("extrn __eq: near");
  outline("extrn __ne: near");
  outline("extrn __le: near");
  outline("extrn __lt: near");
  outline("extrn __ge: near");
  outline("extrn __gt: near");
  outline("extrn __ule: near");
  outline("extrn __ult: near");
  outline("extrn __uge: near");
  outline("extrn __ugt: near");
  outline("extrn __lneg: near");
  outline("extrn __switch: near");
  outline("dw 0"); /* force non-zero code pointers, word alignment */
  toseg(DATASEG);
  outline("dw 0"); /* force non-zero data pointers, word alignment */
  }

/*
** print any assembler stuff needed at the end
*/
trailer()  {
  char *cp;
```

```
  cptr = STARTGLB;
  while(cptr < ENDGLB) {
    if(cptr[IDENT] == FUNCTION && cptr[CLASS] == AUTOEXT)
      external(cptr + NAME, 0, FUNCTION);
    cptr += SYMMAX;
    }
  if((cp = findglb("main")) && cp[CLASS]==STATIC)
    external("_main", 0, FUNCTION);
  toseg(NULL);
  outline("END");
#ifdef DISOPT
    {
    int i, *count;
    printf(";opt   count\n");
    for(i = -1; ++i <= HIGH_SEQ; ) {
      count = seq[i];
      printf("; %2u   %5u\n", i, *count);
      poll(YES);
      }
    }
#endif
  }

/*
** remember where we are in the queue in case we have to back up.
*/
setstage(before, start) int *before, *start; {
  if((*before = snext) == 0)
    snext = stage;
  *start = snext;
  }

/*
** generate code in staging buffer.
*/
gen(pcode, value) int pcode, value; {
  int newcsp;
  switch(pcode) {
    case GETb1pu:
    case GETb1p:
    case GETw1p: gen(MOVE21, 0); break;
    case SUB12:
    case MOD12:
    case MOD12u:
```

```
    case DIV12:
    case DIV12u: gen(SWAP12, 0); break;
    case PUSH1:  csp -= BPW;     break;
    case POP2:   csp += BPW;     break;
    case ADDSP:
    case RETURN: newcsp = value; value -= csp; csp = newcsp;
    }
  if(snext == 0) {
    outcode(pcode, value);
    return;
    }
  if(snext >= slast) {
    error("staging buffer overflow");
    return;
    }
  snext[0] = pcode;
  snext[1] = value;
  snext += 2;
  }

/*
** dump the contents of the queue.
** If start = 0, throw away contents.
** If before != 0, don't dump queue yet.
*/
clearstage(before, start) int *before, *start; {
  if(before) {
    snext = before;
    return;
    }
  if(start) dumpstage();
  snext = 0;
  }

/*
** dump the staging buffer
*/
dumpstage() {
  int i;
  stail = snext;
  snext = stage;
  while(snext < stail) {
    if(optimize) {
      restart:
```

```
      i = -1;
      while(++i <= HIGH_SEQ) if(peep(seq[i])) {
#ifdef DISOPT
        if(isatty(output))
          fprintf(stderr, "                    optimized %2u\n", i);
#endif
        goto restart;
        }
      }
    outcode(snext[0], snext[1]);
    snext += 2;
    }
  }

/*
** change to a new segment
** may be called with NULL, CODESEG, or DATASEG
*/
toseg(newseg) int newseg; {
  if(oldseg == newseg)  return;
  if(oldseg == CODESEG) outline("CODE ENDS");
  else if(oldseg == DATASEG) outline("DATA ENDS");
  if(newseg == CODESEG) {
    outline("CODE SEGMENT PUBLIC");
    outline("ASSUME CS:CODE, SS:DATA, DS:DATA");
    }
  else if(newseg == DATASEG) outline("DATA SEGMENT PUBLIC");
  oldseg = newseg;
  }

/*
** declare entry point
*/
public(ident) int ident;{
  if(ident == FUNCTION)
       toseg(CODESEG);
  else toseg(DATASEG);
  outstr("PUBLIC ");
  outname(ssname);
  newline();
  outname(ssname);
  if(ident == FUNCTION) {
    colon();
    newline();
```

```
    }
  }

/*
** declare external reference
*/
external(name, size, ident) char *name; int size, ident; {
  if(ident == FUNCTION)
       toseg(CODESEG);
  else toseg(DATASEG);
  outstr("EXTRN ");
  outname(name);
  colon();
  outsize(size, ident);
  newline();
  }

/*
** output the size of the object pointed to.
*/
outsize(size, ident) int size, ident; {
  if(size == 1
  && ident != POINTER
  && ident != FUNCTION)      outstr("BYTE");
  else if(ident != FUNCTION) outstr("WORD");
  else                       outstr("NEAR");
  }

/*
** point to following object(s)
*/
point() {
  outline(" DW $+2");
  }

/*
** dump the literal pool
*/
dumplits(size) int size; {
  int j, k;
  k = 0;
  while (k < litptr) {
    poll(1);                    /* allow program interruption */
    if(size == 1)
```

```
        gen(BYTE_, NULL);
    else gen(WORD_, NULL);
    j = 10;
    while(j--) {
      outdec(getint(litq + k, size));
      k += size;
      if(j == 0 || k >= litptr) {
        newline();
        break;
        }
      fputc(',', output);
      }
    }
  }

/*
** dump zeroes for default initial values
*/
dumpzero(size, count) int size, count; {
  if(count > 0) {
    if(size == 1)
        gen(BYTEr0, count);
    else gen(WORDr0, count);
    }
  }

/******************** optimizer functions ***********************/

/*
** Try to optimize sequence at snext in the staging buffer.
*/
peep(seq) int *seq; {
  int *next, *count, *pop, n, skip, tmp, reply;
  char c;
  next = snext;
  count = seq++;
  while(*seq) {
    switch(*seq) {
      case any:   if(next < stail)       break;      return (NO);
      case pfree: if(isfree(PRI, next))  break;      return (NO);
      case sfree: if(isfree(SEC, next))  break;      return (NO);
      case comm:  if(*next & COMMUTES)   break;      return (NO);
      case _pop:  if(pop = getpop(next)) break;      return (NO);
      default:    if(next >= stail || *next != *seq) return (NO);
```

```
      }
    next += 2; ++seq;
    }

  /****** have a match, now optimize it ******/

  *count += 1;
  reply = skip = NO;
  while(*(++seq) || skip) {
    if(skip) {
      if(*seq == 0) skip = NO;
      continue;
      }
    if(*seq >= PCODES) {
      c = *seq & 0xFF;            /* get low byte of command */
      n = c;                      /* and sign extend into n */
      switch(*seq & 0xFF00) {
        case ife:   if(snext[1] != n) skip = YES;  break;
        case ifl:   if(snext[1] >= n) skip = YES;  break;
        case go:    snext += (n<<1);               break;
        case gc:    snext[0] =  snext[(n<<1)];     goto done;
        case gv:    snext[1] =  snext[(n<<1)+1];   goto done;
        case sum:   snext[1] += snext[(n<<1)+1];   goto done;
        case neg:   snext[1] = -snext[1];          goto done;
        case topop: pop[0] = n; pop[1] = snext[1]; goto done;
        case swv:   tmp = snext[1];
                    snext[1] = snext[(n<<1)+1];
                    snext[(n<<1)+1] = tmp;
        done:       reply = YES;
                    break;
        }
      }
    else snext[0] = *seq;         /* set p-code */
    }
  return (reply);
  }

/*
** Is the primary or secondary register free?
** Is it zapped or unused by the p-code at pp
** or a successor?  If the primary register is
** unused by it still may not be free if the
** context uses the value of the expression.
*/
```

```
isfree(reg, pp) int reg, *pp; {
  char *cp;
  while(pp < stail) {
    cp = code[*pp];
    if(*cp & USES & reg) return (NO);
    if(*cp & ZAPS & reg) return (YES);
    pp += 2;
    }
  if(usexpr) return (reg & 001);   /* PRI => NO, SEC => YES at end */
  else       return (YES);
  }

/*
** Get place where the currently pushed value is popped?
** NOTE: Function arguments are not popped, they are
** wasted with an ADDSP.
*/
getpop(next) int *next; {
  char *cp;
  int level;  level = 0;
  while(YES) {
    if(next >= stail)                    /* compiler error */
      return 0;
    if(*next == POP2)
      if(level) --level;
      else return next;                  /* have a matching POP2 */
    else if(*next == ADDSP) {            /* after func call */
      if((level -= (next[1]>>LBPW)) < 0)
        return 0;
      }
    else {
      cp = code[*next];                  /* code string ptr */
      if(*cp & PUSHES) ++level;          /* must be a push */
      }
    next += 2;
    }
  }

/****************** output functions ********************/

colon() {
  fputc(':', output);
  }
```

```
newline() {
  fputc(NEWLINE, output);
  }

/*
** output assembly code.
*/
outcode(pcode, value) int pcode, value; {
  int part, skip, count;
  char *cp, *back;
  part = back = 0;
  skip = NO;
  cp = code[pcode] + 1;          /* skip 1st byte of code string */
  while(*cp) {
    if(*cp == '<') {
      ++cp;                      /* skip to action code */
      if(skip == NO) switch(*cp) {
        case 'm': outname(value+NAME); break; /* mem ref by label */
        case 'n': outdec(value);       break; /* numeric constant */
        case 'l': outdec(litlab);      break; /* current literal label */
        }
      cp += 2;                   /* skip past > */
      }
    else if(*cp == '?') {        /* ?..if value...?...if not value...? */
      switch(++part) {
        case 1: if(value == 0) skip = YES; break;
        case 2: skip = !skip;              break;
        case 3: part = 0; skip = NO;       break;
        }
      ++cp;                      /* skip past ? */
      }
    else if(*cp == '#') {        /* repeat #...# value times */
      ++cp;
      if(back == 0) {
        if((count = value) < 1) {
          while(*cp && *cp++ != '#') ;
          continue;
          }
        back = cp;
        continue;
        }
      if(--count > 0) cp = back;
      else back = 0;
      }
```

```
    else if(skip == NO) fputc(*cp++, output);
    else ++cp;
    }
  }

outdec(number)  int number; {
  int k, zs;
  char c, *q, *r;
  zs = 0;
  k = 10000;
  if(number < 0) {
    number = -number;
    fputc('-', output);
    }
  while (k >= 1) {
    q = 0;
    r = number;
    while(r >= k) {++q; r = r - k;}
    c = q + '0';
    if(c != '0' || k == 1 || zs) {
      zs = 1;
      fputc(c, output);
      }
    number = r;
    k /= 10;
    }
  }

outline(ptr)  char ptr[];  {
  outstr(ptr);
  newline();
  }

outname(ptr) char ptr[]; {
  outstr("_");
  while(*ptr >= ' ') fputc(toupper(*ptr++), output);
  }

outstr(ptr) char ptr[]; {
  poll(1);           /* allow program interruption */
  while(*ptr >= ' ') fputc(*ptr++, output);
  }
```

APPENDIX D

Small C Library Listings

CLIB.H

```
/*
** CLIB.H - Definitions for Small-C library functions.
**
** Copyright 1983  L. E. Payne and J. E. Hendrix
*/

/*
** Misc parameters
*/
#define MAXFILES 20  /* maximum open files */
#define DOSEOF   26  /* DOS end-of-file byte */
#define ARCHIVE  32  /* file archive bit */

/*
** DOS function calls
*/
#define CREATE   60  /* make file */
#define OPEN     61  /* open file */
#define CLOSE    62  /* close file or device */
#define READ     63  /* read from a file */
#define WRITE    64  /* write to a file */
#define DELETE   65  /* delete file */
#define SEEK     66  /* seek within a file */
#define CONTROL  68  /* control device */
#define FORCE    70  /* force use of a handle */
#define RETDOS   76  /* close files and return to DOS */
#define FNDFIL   78  /* find first occurrence of a file */
#define FNDNXT   79  /* find next occurrence of a file */
#define RENAME   86  /* rename file */

/*
** File status bits
*/
#define OPNBIT   1  /* open condition */
```

```
#define EOFBIT   2  /* end-of-file condition */
#define ERRBIT   4  /* error condition */

/*
** File positioning origins
*/
#define FROM_BEG  0  /* from beginning of file */
#define FROM_CUR  1  /* from current position */
#define FROM_END  2  /* from end of file */

/*
** Buffer usage codes
** NULL means the buffer is not used.
*/
#define EMPTY     1  /* buffer is currently empty */
#define IN        2  /* buffer is currently holding input data */
#define OUT       3  /* buffer is currently holding output data */

/*
** ASCII characters
*/
#define ABORT    3
#define RUB      8
#define PAUSE   19
#define WIPE    24
#define DEL    127
```

ABS.C

```
/*
** abs - returns absolute value of nbr
*/
abs(nbr)  int nbr; {
  if(nbr < 0) return (-nbr);
  return (nbr);
  }
```

ATOI.C

```
/*
** atoi(s) - convert s to integer.
*/
atoi(s) char *s; {
```

```
  int sign, n;
  while(isspace(*s)) ++s;
  sign = 1;
  switch(*s) {
    case '-': sign = -1;
    case '+': ++s;
    }
  n = 0;
  while(isdigit(*s)) n = 10 * n + *s++ - '0';
  return (sign * n);
  }
```

ATOIB.C

```
/*
** atoib(s,b) - Convert s to "unsigned" integer in base b.
**              NOTE: This is a non-standard function.
*/
atoib(s, b) char *s; int b; {
  int n, digit;
  n = 0;
  while(isspace(*s)) ++s;
  while((digit = (127 & *s++)) >= '0') {
    if(digit >= 'a')      digit -= 87;
    else if(digit >= 'A') digit -= 55;
    else                  digit -= '0';
    if(digit >= b) break;
    n = b * n + digit;
    }
  return (n);
  }
```

AUXBUF.C

```
#include "stdio.h"
#include "clib.h"
extern int
  _bufsiz[MAXFILES],  /* size of buffer */
  _bufptr[MAXFILES];  /* aux buffer address */

/*
** auxbuf - allocate an auxiliary input buffer for fd
**   fd = file descriptor of an open file
```

```
** size = size of buffer to be allocated
** Returns NULL on success, else ERR.
** Note: Ungetc() still works.
**       A 2nd call allocates a new buffer replacing old one.
**       If fd is a device, buffer is allocated but ignored.
**       Buffer stays allocated when fd is closed or new one is allocated.
**       May be used on a closed fd.
*/
auxbuf(fd, size) int fd; char *size; {   /* fake unsigned */
  if(!size || avail(NO) < size) return (ERR);
  _bufptr[fd] = malloc(size);
  _bufsiz[fd] = size;
  _empty(fd, NO);
  return (NULL);
  }
```

AVAIL.C

```
extern char *_memptr;
/*
** Return the number of bytes of available memory.
** In case of a stack overflow condition, if 'abort'
** is non-zero the program aborts with an 'S' clue,
** otherwise zero is returned.
*/
avail(abort) int abort; {
  char x;
  if(&x < _memptr) {
    if(abort) exit(1);
    return (0);
    }
  return (&x - _memptr);
  }
```

BSEEK.C

```
#include "stdio.h"
#include "clib.h"
extern int _nextc[], _bufuse[];
/*
** Position fd to the character in file indicated by "offset."
** "Offset" is the address of a long integer or array of two
** integers containing the offset, low word first.
```

```
**
**     BASE      OFFSET-RELATIVE-TO
**       0       beginning of file
**       1       current byte in file
**       2       end of file (minus offset)
**
** Returns NULL on success, else EOF.
*/
bseek(fd, offset, base) int fd, offset[], base; {
  int hi, lo;
  if(!_mode(fd) || !_bufuse[fd]) return (EOF);
  if(_adjust(fd)) return (EOF);
  lo = offset[0];
  hi = offset[1];
  if(!_seek(base, fd, &hi, &lo)) return (EOF);
  _nextc[fd] = EOF;
  _clreof(fd);
  return (NULL);
  }
```

BTELL.C

```
#include "stdio.h"
#include "clib.h"
extern int _bufuse[];
/*
** Retrieve offset to next character in fd.
** "Offset" must be the address of a long int or
** a 2-element int array.  The offset is placed in
** offset in low, high order.
*/
btell(fd, offset) int fd, offset[]; {
  if(!_mode(fd) || !_bufuse[fd]) return (EOF);
  if(_adjust(fd)) return (EOF);
  offset[0] = offset[1] = 0;
  _seek(FROM_CUR, fd, offset+1, offset);
  return (NULL);
  }
```

CALL.ASM

```
;
; Small-C Run Time Library for MS/PC-DOS
;
        extrn   __main: near
        extrn    _exit: near
        extrn   __memptr: word

data    segment public
        dw      1
data    ends

stack   segment stack
        dw      32 dup(?)
stack   ends

code    segment public
        assume  cs:code
start:
        mov     ax,data         ; set data segment for program
        mov     ds,ax
        mov     ax,es:[2]       ; paragraphs of memory on system
        sub     ax,data         ; paragraphs beyond code segment
        cmp     ah,10h          ; more than 64K?
        jb      start_1         ; no
        mov     ax,1000h        ; only use 64K
start_1:
        mov     cl,4
        shl     ax,cl           ; byte offset to end of data/free/stack
        cli                     ; disable interrupts
        mov     bx,ds
        mov     ss,bx           ; make data and stack segments coincide
        mov     sp,ax           ; top of stack = end of data/free/stack
        push    ax              ; force sp non-zero (if 64K used)
        sti                     ; reenable interrupts
        mov     ax,stack        ; paragraph following data
        sub     ax,data         ; number of data paragraphs
        shl     ax,cl           ; number of data bytes (offset to
free/stack)
        mov     bx,ax
        inc     bh              ; adjust for minimum stack space
        cmp     bx,sp           ; enough memory?
        jb      start_2         ; yes
        mov     ax,1            ; no, terminate with exit code 1
```

```
        push    ax
        call    _exit
start_2:
        mov     __memptr,ax     ; set memory allocation pointer
;
; ------- release unused memory -------
; --- cannot run debug with this code ---
;        mov     bx,sp
;        mov     ah,4AH
;        int     21H
; ------------------------
;
; make sure that es -> psp, because __main requires it
;
        jmp     __main          ; __main never returns

        public  _ccargc
_ccargc:
        mov     al,cl
        xor     ah,ah
        ret

;
; Test if Secondary (BX) <oper> Primary (AX)
;
compare macro   name, cond
        public  __&name
__&name:
        cmp     ax,bx
        j&cond  true
        xor     ax,ax   ; returns zero
        ret
        endm
;
        compare ult,a
        compare ugt,b
        compare ule,ae
        compare uge,be
        compare eq,e
        compare ne,ne
        compare lt,g
        compare gt,l
        compare le,ge
        compare ge,le
```

```
;
; Logical Negate of Primary
;
        public  __lneg
__lneg:
        or      ax,ax
        jnz     false
true:   mov     ax,1    ; returns one
        ret
false:  xor     ax,ax   ; returns zero
        ret
;
;
; execute "switch" statement
;
;  ax  =  switch value
; (sp) -> switch table
;         dw addr1, value1
;         dw addr2, value2
;         ...
;         dw 0
;        [jmp default]
;         continuation
;
        public  __switch
__switch:
        pop     bx              ; bx -> switch table
        jmp     short skip      ; skip the pre-increment
loop:
        add     bx,4
skip:   mov     cx,cs:[bx]
        jcxz    default         ; end of table - jump out
        cmp     ax,cs:[bx+2]
        jnz     loop
        jmp     cx              ; match - jump to case
default:
        inc     bx
        inc     bx
        jmp     bx              ; jump to default/continuation
;
; dummy entry point to resolve the external reference _LINK
; which is no longer generated by Small-C but which exists in
; library modules and .OBJ files compiled by earlier versions
```

```
; of Small-C
        public  __link
__link: ret

code ends
        end     start
```

CALLOC.C

```
#include "stdio.h"
/*
** Cleared-memory allocation of n items of size bytes.
** n      = Number of items to allocate space for.
** size   = Size of the items in bytes.
** Returns the address of the allocated block,
** else NULL for failure.
*/
calloc(n, size) unsigned n, size; {
  return (_alloc(n*size, YES));
  }
```

CLEARERR.C

```
#include "stdio.h"
#include "clib.h"
extern int _status[];
/*
** Clear error status for fd.
*/
clearerr(fd) int fd; {
  if(_mode(fd)) _clrerr(fd);
  }
```

CSEEK.C

```
#include "stdio.h"
#include "clib.h"
extern int _nextc[], _bufuse[];
/*
** Position fd to the 128-byte record indicated by
** "offset" relative to the point indicated by "base."
**
```

```
**      BASE      OFFSET-RELATIVE-TO
**        0       first record
**        1       current record
**        2       end of file (last record + 1)
**                (offset should be minus)
**
** Returns NULL on success, else EOF.
*/
cseek(fd, offset, base) int fd, offset, base; {
  int newrec, oldrec, hi, lo;
  if(!_mode(fd) || !_bufuse[fd]) return (EOF);
  if(_adjust(fd)) return (EOF);
  switch (base) {
     case 0: newrec = offset;
             break;
     case 1: oldrec = ctell(fd);
             goto calc;
     case 2: hi = lo = 0;
             if(!_seek(FROM_END, fd, &hi, &lo)) return (EOF);
             oldrec = ((lo >> 7) & 511) | (hi << 9);
     calc:
             newrec = oldrec + offset;
             break;
    default: return (EOF);
    }
  lo = (newrec << 7);       /* convert newrec to long int */
  hi = (newrec >> 9) & 127;
  if(!_seek(FROM_BEG, fd, &hi, &lo)) return (EOF);
  _nextc[fd] = EOF;
  _clreof(fd);
  return (NULL);
  }

/*
** Position fd to the character indicated by
** "offset" within current 128-byte record.
** Must be on record boundary.
**
** Returns NULL on success, else EOF.
*/
cseekc(fd, offset) int fd, offset; {
  int hi, lo;
  if(!_mode(fd) || isatty(fd) ||
     ctellc(fd) || offset < 0 || offset > 127) return (EOF);
```

```
  hi = 0; lo = offset;
  if(!_seek(FROM_CUR, fd, &hi, &lo)) return (EOF);
  return (NULL);
  }
```

CSYSLIB.C

```
/*
** CSYSLIB - System-Level Library Functions
*/

#include "stdio.h"
#include "clib.h"

/*
****************** System Variables ********************
*/

int
  _cnt=1,              /* arg count for main */
  _vec[20],            /* arg vectors for main */
  _status[MAXFILES] = {OPNBIT, OPNBIT, OPNBIT, OPNBIT, OPNBIT},
  _cons  [MAXFILES],   /* fast way to tell if file is the console */
  _nextc [MAXFILES] = {EOF, EOF, EOF, EOF, EOF},
  _bufuse[MAXFILES],   /* current buffer usage: NULL, EMPTY, IN, OUT */
  _bufsiz[MAXFILES],   /* size of buffer */
  _bufptr[MAXFILES],   /* aux buffer address */
  _bufnxt[MAXFILES],   /* address of next byte in buffer */
  _bufend[MAXFILES],   /* address of end-of-data in buffer */
  _bufeof[MAXFILES];   /* true if current buffer ends file */

char
 *_memptr,            /* pointer to free memory. */
  _arg1[]="*";        /* first arg for main */

/*
*************** System-Level Functions *****************
*/

/*
**  Process command line, allocate default buffer to each fd,
**  execute main(), and exit to DOS.  Must be executed with es=psp.
**  Small default buffers are allocated because a high price is paid for
```

```
**  byte-by-byte calls to DOS.  Tests gave these results for a simple
**  copy program:
**
**          chunk size       copy time in seconds
**              1                    36
**              5                    12
**             25                     6
**             50                     6
*/
_main() {
  int fd;
  _parse();
  for(fd = 0; fd < MAXFILES; ++fd) auxbuf(fd, 32);
  if(!isatty(stdin))  _bufuse[stdin]  = EMPTY;
  if(!isatty(stdout)) _bufuse[stdout] = EMPTY;
  main(_cnt, _vec);
  exit(0);
  }

/*
** Parse command line and setup argc and argv.
** Must be executed with es == psp
*/
_parse() {
  char *ptr;
#asm
  mov     cl,es:[80h]  ; get parameter string length
  mov     ch,0
  push    cx           ; save it
  inc     cx
  push    cx           ; 1st __alloc() arg
  mov     ax,1
  push    ax           ; 2nd __alloc() arg
  call    __alloc      ; allocate zeroed memory for args
  add     sp,4
  mov     [bp-2],ax    ; ptr = addr of allocated memory
  pop     cx
  push    es           ; exchange
  push    ds           ; es         (source)
  pop     es           ;    and
  pop     ds           ;        ds  (destination)
  mov     si,81h       ; source offset
  mov     di,[bp-2]    ; destination offset
  rep     movsb        ; move string
```

```
  mov     al,0
  stosb                   ; terminate with null byte
  push    es
  pop     ds              ; restore ds
#endasm
  _vec[0]=_arg1;          /* first arg = "*" */
  while (*ptr) {
    if(isspace(*ptr)) {++ptr; continue;}
    if(_cnt < 20) _vec[_cnt++] = ptr;
    while(*ptr) {
      if(isspace(*ptr)) {*ptr = NULL; ++ptr; break;}
      ++ptr;
      }
    }
  }

/*
** Open file on specified fd.
*/
_open(fn, mode, fd) char *fn, *mode; int *fd; {
  int rw, tfd;
  switch(mode[0]) {
    case 'r': {
      if(mode[1] == '+') rw = 2; else rw = 0;
      if ((tfd = _bdos2((OPEN<<8)|rw, NULL, NULL, fn)) < 0) return (NO);
      break;
      }
    case 'w': {
      if(mode[1] == '+') rw = 2; else rw = 1;
    create:
      if((tfd = _bdos2((CREATE<<8), NULL, ARCHIVE, fn)) < 0) return (NO);
      _bdos2(CLOSE<<8, tfd, NULL, NULL);
      if((tfd = _bdos2((OPEN<<8)|rw, NULL, NULL, fn)) < 0) return (NO);
      break;
      }
    case 'a': {
      if(mode[1] == '+') rw = 2; else rw = 1;
      if((tfd = _bdos2((OPEN<<8)|rw, NULL, NULL, fn)) < 0) goto create;
      if(_bdos2((SEEK<<8)|FROM_END, tfd, NULL, 0) < 0) return (NO);
      break;
      }
    default: return (NO);
    }
  _empty(tfd, YES);
```

```
  if(isatty(tfd)) _bufuse[tfd] = NULL;
  *fd = tfd;
  _cons  [tfd] = NULL;
  _nextc [tfd] = EOF;
  _status[tfd] = OPNBIT;
  return (YES);
  }

/*
** Binary-stream input of one byte from fd.
*/
_read(fd) int fd; {
  unsigned char ch;
  if(_nextc[fd] != EOF) {
    ch = _nextc[fd];
    _nextc[fd] = EOF;
    return (ch);
    }
  if(iscons(fd))  return (_getkey());
  if(_bufuse[fd]) return(_readbuf(fd));
  switch(_bdos2(READ<<8, fd, 1, &ch)) {
     case 1:  return (ch);
     case 0: _seteof(fd); return (EOF);
    default: _seterr(fd); return (EOF);
    }
  }

/*
** Fill buffer if necessary, and return next byte.
*/
_readbuf(fd) int fd; {
  int got, chunk;
  char *ptr, *max;
  if(_bufuse[fd] == OUT && _flush(fd)) return (EOF);
  while(YES) {
    ptr = _bufnxt[fd];
    if(ptr < _bufend[fd]) {++_bufnxt[fd]; return (*ptr);}
    if(_bufeof[fd]) {_seteof(fd); return (EOF);}
    max = (ptr = _bufend[fd] = _bufptr[fd]) + _bufsiz[fd];
    do {          /* avoid DMA problem on physical 64K boundary */
      if((max - ptr) < 512) chunk = max - ptr;
      else                  chunk = 512;
      ptr += (got = _bdos2(READ<<8, fd, chunk, ptr));
      if(got < chunk) {_bufeof[fd] = YES; break;}
```

```
      } while(ptr < max);
    _bufend[fd] = ptr;
    _bufnxt[fd] = _bufptr[fd];
    _bufuse[fd] = IN;
    }
  }

/*
** Binary-Stream output of one byte to fd.
*/
_write(ch, fd) int ch, fd; {
  if(_bufuse[fd]) return(_writebuf(ch, fd));
  if(_bdos2(WRITE<<8, fd, 1, &ch) != 1) {
    _seterr(fd);
    return (EOF);
    }
  return (ch);
  }

/*
** Empty buffer if necessary, and store ch in buffer.
*/
_writebuf(ch, fd) int ch, fd; {
  char *ptr;
  if(_bufuse[fd] == IN && _backup(fd)) return (EOF);
  while(YES) {
    ptr = _bufnxt[fd];
    if(ptr < (_bufptr[fd] + _bufsiz[fd])) {
      *ptr = ch;
      ++_bufnxt[fd];
      _bufuse[fd] = OUT;
      return (ch);
      }
    if(_flush(fd)) return (EOF);
    }
  }

/*
** Flush buffer to DOS if dirty buffer.
** Reset buffer pointers in any case.
*/
_flush(fd) int fd; {
  int i, j, k, chunk;
  if(_bufuse[fd] == OUT) {
```

```
    i = _bufnxt[fd] - _bufptr[fd];
    k = 0;
    while(i > 0) {      /* avoid DMA problem on physical 64K boundary */
      if(i < 512) chunk = i;
      else        chunk = 512;
      k += (j = _bdos2(WRITE<<8, fd, chunk, _bufptr[fd] + k));
      if(j < chunk) {_seterr(fd); return (EOF);}
      i -= j;
      }
    }
  _empty(fd, YES);
  return (NULL);
  }

/*
** Adjust DOS file position to current point.
*/
_adjust(fd) int fd; {
  if(_bufuse[fd] == OUT) return (_flush(fd));
  if(_bufuse[fd] == IN ) return (_backup(fd));
  }

/*
** Backup DOS file position to current point.
*/
_backup(fd) int fd; {
  int hi, lo;
  if(lo = _bufnxt[fd] - _bufend[fd]) {
    hi = -1;
    if(!_seek(FROM_CUR, fd, &hi, &lo)) {
      _seterr(fd);
      return (EOF);
      }
    }
  _empty(fd, YES);
  return (NULL);
  }

/*
** Set buffer controls to empty status.
*/
_empty(fd, mt) int fd, mt; {
  _bufnxt[fd] = _bufend[fd] = _bufptr[fd];
  _bufeof[fd] = NO;
```

```
  if(mt) _bufuse[fd] = EMPTY;
  }

/*
** Return fd's open mode, else NULL.
*/
_mode(fd) char *fd; {
  if(fd < MAXFILES) return (_status[fd]);
  return (NULL);
  }

/*
** Set eof status for fd and
*/
_seteof(fd) int fd; {
  _status[fd] |= EOFBIT;
  }

/*
** Clear eof status for fd.
*/
_clreof(fd) int fd; {
  _status[fd] &=  EOFBIT;
  }

/*
** Set error status for fd.
*/
_seterr(fd) int fd; {
  _status[fd] |= ERRBIT;
  }

/*
** Clear error status for fd.
*/
_clrerr(fd) int fd; {
  _status[fd] &=  ERRBIT;
  }

/*
** Allocate n bytes of (possibly zeroed) memory.
** Entry: n = Size of the items in bytes.
**    clear = "true" if clearing is desired.
** Returns the address of the allocated block of memory
```

```
** or NULL if the requested amount of space is not available.
*/
_alloc(n, clear) unsigned n, clear; {
  char *oldptr;
  if(n < avail(YES)) {
    if(clear) pad(_memptr, NULL, n);
    oldptr = _memptr;
    _memptr += n;
    return (oldptr);
    }
  return (NULL);
  }

/*
** Issue extended BDOS function and return result.
** Entry: ax = function code and sub-function
**        bx, cx, dx = other parameters
*/
_bdos2(ax, bx, cx, dx) int ax, bx, cx, dx; {
#asm
  push bx          ; preserve secondary register
  mov  dx,[bp+4]
  mov  cx,[bp+6]
  mov  bx,[bp+8]
  mov  ax,[bp+10] ; load DOS function number
  int  21h         ; call bdos
  jnc  __bdos21    ; no error
  neg  ax          ; make error code negative
__bdos21:
  pop  bx          ; restore secondary register
#endasm
  }

/*
** Issue LSEEK call
*/
_seek(org, fd, hi, lo) int org, fd, hi, lo; {
#asm
  push bx          ; preserve secondary register
  mov  bx,[bp+4]
  mov  dx,[bx]     ; get lo part of destination
  mov  bx,[bp+6]
  mov  cx,[bx]     ; get hi part of destination
  mov  bx,[bp+8]   ; get file descriptor
```

```
  mov  al,[bp+10] ; get origin code for seek
  mov  ah,42h     ; move-file-pointer function
  int  21h        ; call bdos
  jnc  __seek1    ; error?
  xor  ax,ax      ; yes, return false
  jmp  __seek2
__seek1:          ; no, set hi and lo
  mov  bx,[bp+4]  ; get address of lo
  mov  [bx],ax    ; store low part of new position
  mov  bx,[bp+6]  ; get address of hi
  mov  [bx],dx    ; store high part of new position
  mov  ax,1       ; return true
__seek2:
  pop  bx         ; restore secondary register
#endasm
  }

/*
** Test for keyboard input
*/
_hitkey() {
#asm
  mov  ah,1       ; sub-service = test keyboard
  int  16h        ; call bdos keyboard services
  jnz  __hit1
  xor ax,ax       ; nothing there, return false
  jmp  __hit2
__hit1:
  mov  ax,1       ; character ready, return true
__hit2:
#endasm
  }

/*
** Return next keyboard character
*/
_getkey() {
#asm
  mov  ah,0       ; sub-service = read keyboard
  int  16h        ; call bdos keyboard services
  or   al,al      ; special character?
  jnz  __get2     ; no
  mov  al,ah      ; yes, move it to al
  cmp  al,3       ; ctl-2 (simulated null)?
```

```
  jne  __get1      ; no
  xor  al,al       ; yes, report zero
  jmp  __get2
__get1:
  add  al,113      ; offset to range 128-245
__get2:
  xor  ah,ah       ; zero ah
#endasm
  }
```

CTELL.C

```
#include "stdio.h"
#include "clib.h"
extern int _bufuse[];
/*
** Return offset to current 128-byte record.
*/
ctell(fd) int fd; {
  int hi, lo;
  if(!_mode(fd) || !_bufuse[fd]) return (-1);
  if(_adjust(fd)) return (-1);
  hi = lo = 0;
  _seek(FROM_CUR, fd, &hi, &lo);
  return ((hi << 9) | ((lo >> 7) & 511));
  }

/*
** Return offset to next byte in current 128-byte record.
*/
ctellc(fd) int fd; {
  int hi, lo;
  if(!_mode(fd) || !_bufuse[fd]) return (-1);
  if(_adjust(fd)) return (-1);
  hi = lo = 0;
  _seek(FROM_CUR, fd, &hi, &lo);
  return (lo & 127);
  }
```

DTOI.C

```
#include "stdio.h"
/*
```

```
** dtoi - convert signed decimal string to integer nbr
**         returns field length, else ERR on error
*/
dtoi(decstr, nbr)  char *decstr;  int *nbr;  {
  int len, s;
  if((*decstr)=='-') {s=1; ++decstr;} else s=0;
  if((len=utoi(decstr, nbr))<0) return ERR;
  if(*nbr<0) return ERR;
  if(s) {*nbr = -*nbr; return ++len;} else return len;
  }
```

EXIT.C

```
#include "stdio.h"
#include "clib.h"
/*
** Close all open files and exit to DOS.
** Entry: ec = exit code.
*/
exit(ec) int ec; {
  int fd;  char str[4];
  ec &= 255;
  if(ec) {
    left(itou(ec, str, 4));
    fputs("Exit Code: ", stderr);
    fputs(str, stderr);
    fputs("\n", stderr);
    }
  for(fd = 0; fd < MAXFILES; ++fd) fclose(fd);
  _bdos2((RETDOS<<8)|ec, NULL, NULL, NULL);
#asm
_abort: jmp    _exit
        public _abort
#endasm
  }
```

FCLOSE.C

```
#include "stdio.h"
#include "clib.h"
/*
** Close fd
** Entry: fd = file descriptor for file to be closed.
```

```
** Returns NULL for success, otherwise ERR
*/
extern int _status[];
fclose(fd) int fd; {
  if(!_mode(fd) || _flush(fd)) return (ERR);
  if(_bdos2(CLOSE<<8, fd, NULL, NULL) == -6) return (ERR);
  return (_status[fd] = NULL);
  }
```

FEOF.C

```
#include "clib.h"
extern int _status[];
/*
** Test for end-of-file status.
** Entry: fd = file descriptor
** Returns non-zero if fd is at eof, else zero.
*/
feof(fd) int fd; {
  return (_status[fd] & EOFBIT);
  }
```

FERROR.C

```
#include "stdio.h"
#include "clib.h"
extern _status[];
/*
** Test for error status on fd.
*/
ferror(fd) int fd; {
  return (_status[fd] & ERRBIT);
  }
```

FGETC.C

```
#include "stdio.h"
#include "clib.h"

extern int _nextc[];

/*
```

```
** Character-stream input of one character from fd.
** Entry: fd = File descriptor of pertinent file.
** Returns the next character on success, else EOF.
*/
fgetc(fd) int fd; {
  int ch;                   /* must be int so EOF will flow through */
  if(_nextc[fd] != EOF) {   /* an ungotten byte pending? */
    ch = _nextc[fd];
    _nextc[fd] = EOF;
    return (ch & 255);      /* was cooked the first time */
    }
  while(1) {
    ch = _read(fd);
    if(iscons(fd)) {
      switch(ch) {          /* extra console cooking */
        case ABORT:  exit(2);
        case    CR: _write(CR, stderr); _write(LF, stderr); break;
        case   DEL:  ch = RUB;
        case   RUB:
        case  WIPE:  break;
        default:    _write(ch, stderr);
        }
      }
    switch(ch) {            /* normal cooking */
          default:  return (ch);
      case DOSEOF: _seteof(fd); return (EOF);
      case     CR:  return ('\n');
      case     LF:
      }
    }
  }
#asm
_getc:  jmp     _fgetc
        public  _getc
#endasm
```

FGETS.C

```
#include "stdio.h"
#include "clib.h"
/*
** Gets an entire string (including its newline
** terminator) or size-1 characters, whichever comes
```

```
** first. The input is terminated by a null character.
** Entry: str  = Pointer to destination buffer.
**        size = Size of the destination buffer.
**        fd   = File descriptor of pertinent file.
** Returns str on success, else NULL.
*/
fgets(str, size, fd) char *str; unsigned size, fd; {
  return (_gets(str, size, fd, 1));
  }

/*
** Gets an entire string from stdin (excluding its newline
** terminator) or size-1 characters, whichever comes
** first. The input is terminated by a null character.
** The user buffer must be large enough to hold the data.
** Entry: str  = Pointer to destination buffer.
** Returns str on success, else NULL.
*/
gets(str) char *str; {
  return (_gets(str, 32767, stdin, 0));
  }

_gets(str, size, fd, nl) char *str; unsigned size, fd, nl; {
  int backup; char *next;
  next = str;
  while(--size > 0) {
    switch (*next = fgetc(fd)) {
      case  EOF: *next = NULL;
                 if(next == str) return (NULL);
                 return (str);
      case '\n': *(next + nl) = NULL;
                 return (str);
      case  RUB: if(next > str) backup = 1; else backup = 0;
                 goto backout;
      case WIPE: backup = next - str;
        backout: if(iscons(fd)) {
                   ++size;
                   while(backup--) {
                     fputs("\b \b", stderr);
                     --next; ++size;
                     }
                   continue;
                   }
        default: ++next;
```

```
      }
    }
  *next = NULL;
  return (str);
  }
```

FOPEN.C

```
#include "stdio.h"
#include "clib.h"
/*
** Open file indicated by fn.
** Entry: fn   = Null-terminated DOS file name.
**        mode = "a"  - append
**               "r"  - read
**               "w"  - write
**               "a+" - append update
**               "r+" - read   update
**               "w+" - write  update
** Returns a file descriptor on success, else NULL.
*/
fopen(fn, mode) char *fn, *mode; {
  int fd;
  if(!_open(fn, mode, &fd)) return (NULL);
  return (fd);
  }
```

FPRINTF.C

```
#include "stdio.h"
/*
** fprintf(fd, ctlstring, arg, arg, ...) - Formatted print.
** Operates as described by Kernighan & Ritchie.
** b, c, d, o, s, u, and x specifications are supported.
** Note: b (binary) is a non-standard extension.
*/
fprintf(argc) int argc; {
  int *nxtarg;
  nxtarg = CCARGC() + &argc;
  return(_print(*(-nxtarg), -nxtarg));
  }

/*
```

```
** printf(ctlstring, arg, arg, ...) - Formatted print.
** Operates as described by Kernighan & Ritchie.
** b, c, d, o, s, u, and x specifications are supported.
** Note: b (binary) is a non-standard extension.
*/
printf(argc) int argc; {
  return(_print(stdout, CCARGC() + &argc - 1));
  }

/*
** _print(fd, ctlstring, arg, arg, ...)
** Called by fprintf() and printf().
*/
_print(fd, nxtarg) int fd, *nxtarg; {
  int  arg, left, pad, cc, len, maxchr, width;
  char *ctl, *sptr, str[17];
  cc = 0;
  ctl = *nxtarg--;
  while(*ctl) {
    if(*ctl!='%') {fputc(*ctl++, fd); ++cc; continue;}
    else ++ctl;
    if(*ctl=='%') {fputc(*ctl++, fd); ++cc; continue;}
    if(*ctl=='-') {left = 1; ++ctl;} else left = 0;
    if(*ctl=='0') pad = '0'; else pad = ' ';
    if(isdigit(*ctl)) {
      width = atoi(ctl++);
      while(isdigit(*ctl)) ++ctl;
      }
    else width = 0;
    if(*ctl=='.') {
      maxchr = atoi(++ctl);
      while(isdigit(*ctl)) ++ctl;
      }
    else maxchr = 0;
    arg = *nxtarg--;
    sptr = str;
    switch(*ctl++) {
      case 'c': str[0] = arg; str[1] = NULL; break;
      case 's': sptr = arg;        break;
      case 'd': itoa(arg,str);     break;
      case 'b': itoab(arg,str,2);  break;
      case 'o': itoab(arg,str,8);  break;
      case 'u': itoab(arg,str,10); break;
      case 'x': itoab(arg,str,16); break;
```

```
      default:  return (cc);
      }
    len = strlen(sptr);
    if(maxchr && maxchr<len) len = maxchr;
    if(width>len) width = width - len; else width = 0;
    if(!left) while(width--) {fputc(pad,fd); ++cc;}
    while(len--) {fputc(*sptr++,fd); ++cc; }
    if(left) while(width--) {fputc(pad,fd); ++cc;}
    }
  return(cc);
  }
```

FPUTC.C

```
#include "stdio.h"
#include "clib.h"
extern int _status[];
/*
** Character-stream output of a character to fd.
** Entry: ch = Character to write.
**        fd = File descriptor of perinent file.
** Returns character written on success, else EOF.
*/
fputc(ch, fd) int ch, fd; {
  switch(ch) {
    case  EOF: _write(DOSEOF, fd); break;
    case '\n': _write(CR, fd); _write(LF, fd); break;
      default: _write(ch, fd);
    }
  if(_status[fd] & ERRBIT) return (EOF);
  return (ch);
  }
#asm
_putc:  jmp     _fputc
        public  _putc
#endasm
```

FPUTS.C

```
#include "stdio.h"
#include "clib.h"
/*
** Write a string to fd.
```

```
** Entry: string = Pointer to null-terminated string.
**        fd     = File descriptor of pertinent file.
*/
fputs(string, fd) char *string; int fd; {
  while(*string) fputc(*string++, fd) ;
  }
```

FREAD.C

```
#include "clib.h"
extern int _status[];
/*
** Item-stream read from fd.
** Entry: buf = address of target buffer
**         sz = size of items in bytes
**          n = number of items to read
**         fd = file descriptor
** Returns a count of the items actually read.
** Use feof() and ferror() to determine file status.
*/
fread(buf, sz, n, fd) unsigned char *buf; unsigned sz, n, fd; {
  return (read(fd, buf, n*sz)/sz);
  }

/*
** Binary-stream read from fd.
** Entry:  fd = file descriptor
**        buf = address of target buffer
**          n = number of bytes to read
** Returns a count of the bytes actually read.
** Use feof() and ferror() to determine file status.
*/
read(fd, buf, n) unsigned fd, n; unsigned char *buf; {
  unsigned cnt;
  cnt = 0;
  while(n--) {
    *buf++ = _read(fd);
    if(_status[fd] & (ERRBIT | EOFBIT)) break;
    ++cnt;
    }
  return (cnt);
  }
```

FREE.C

```
extern char *_memptr;
/*
** free(ptr) - Free previously allocated memory block.
** Memory must be freed in the reverse order from which
** it was allocated.
** ptr    = Value returned by calloc() or malloc().
** Returns ptr if successful or NULL otherwise.
*/
free(ptr) char *ptr; {
   return (_memptr = ptr);
   }
#asm
_cfree: jmp     _free
        public  _cfree
#endasm
```

FREOPEN.C

```
#include <stdio.h>
#include "clib.h"
/*
** Close previously opened fd and reopen it.
** Entry: fn   = Null-terminated DOS file name.
**        mode = "a"  - append
**               "r"  - read
**               "w"  - write
**               "a+" - append update
**               "r+" - read   update
**               "w+" - write  update
**        fd   = File descriptor of pertinent file.
** Returns the original fd on success, else NULL.
*/
extern int _status[];
freopen(fn, mode, fd) char *fn, *mode; int fd; {
  int tfd;
  if(fclose(fd)) return (NULL);
  if(!_open(fn, mode, &tfd)) return (NULL);
  if(fd != tfd) {
    if(_bdos2(FORCE<<8, tfd, fd, NULL) < 0) return (NULL);
    _status[fd] = _status[tfd];
    _status[tfd] = 0;       /* leaves DOS using two handles */
    }
```

```
  return (fd);
  }
```

FSCANF.C

```
#include "stdio.h"
/*
** fscanf(fd, ctlstring, arg, arg, ...) - Formatted read.
** Operates as described by Kernighan & Ritchie.
** b, c, d, o, s, u, and x specifications are supported.
** Note: b (binary) is a non-standard extension.
*/
fscanf(argc) int argc; {
  int *nxtarg;
  nxtarg = CCARGC() + &argc;
  return (_scan(*(-nxtarg), -nxtarg));
  }

/*
** scanf(ctlstring, arg, arg, ...) - Formatted read.
** Operates as described by Kernighan & Ritchie.
** b, c, d, o, s, u, and x specifications are supported.
** Note: b (binary) is a non-standard extension.
*/
scanf(argc) int argc; {
  return (_scan(stdin, CCARGC() + &argc - 1));
  }

/*
** _scan(fd, ctlstring, arg, arg, ...) - Formatted read.
** Called by fscanf() and scanf().
*/
_scan(fd,nxtarg) int fd, *nxtarg; {
  char *carg, *ctl;
  unsigned u;
  int  *narg, wast, ac, width, ch, cnv, base, ovfl, sign;
  ac = 0;
  ctl = *nxtarg--;
  while(*ctl) {
    if(isspace(*ctl)) {++ctl; continue;}
    if(*ctl++ != '%') continue;
    if(*ctl == '*') {narg = carg = &wast; ++ctl;}
    else            narg = carg = *nxtarg--;
```

```
  ctl += utoi(ctl, &width);
  if(!width) width = 32767;
  if(!(cnv = *ctl++)) break;
  while(isspace(ch = fgetc(fd))) ;
  if(ch == EOF) {if(ac) break; else return(EOF);}
  ungetc(ch,fd);
  switch(cnv) {
    case 'c':
      *carg = fgetc(fd);
      break;
    case 's':
      while(width--) {
        if((*carg = fgetc(fd)) == EOF) break;
        if(isspace(*carg)) break;
        if(carg != &wast) ++carg;
        }
      *carg = 0;
      break;
    default:
      switch(cnv) {
        case 'b': base =  2; sign = 1; ovfl = 32767; break;
        case 'd': base = 10; sign = 0; ovfl =  3276; break;
        case 'o': base =  8; sign = 1; ovfl =  8191; break;
        case 'u': base = 10; sign = 1; ovfl =  6553; break;
        case 'x': base = 16; sign = 1; ovfl =  4095; break;
        default:  return (ac);
        }
      *narg = u = 0;
      while(width-- && !isspace(ch=fgetc(fd)) && ch!=EOF) {
        if(!sign)
          if(ch == '-') {sign = -1; continue;}
          else sign = 1;
        if(ch < '0') return (ac);
        if(ch >= 'a')      ch -= 87;
        else if(ch >= 'A') ch -= 55;
        else               ch -= '0';
        if(ch >= base || u > ovfl) return (ac);
        u = u * base + ch;
        }
      *narg = sign * u;
    }
  ++ac;
  }
return (ac);
```

```
  }
```

FWRITE.C

```
#include "clib.h"
extern int _status[];
/*
** Item-stream write to fd.
** Entry: buf = address of source buffer
**          sz = size of items in bytes
**           n = number of items to write
**          fd = file descriptor
** Returns a count of the items actually written or
** zero if an error occurred.
** May use ferror(), as always, to detect errors.
*/
fwrite(buf, sz, n, fd) unsigned char *buf; unsigned sz, n, fd; {
  if(write(fd, buf, n*sz) == -1) return (0);
  return (n);
  }

/*
** Binary-stream write to fd.
** Entry:  fd = file descriptor
**        buf = address of source buffer
**          n = number of bytes to write
** Returns a count of the bytes actually written or
** -1 if an error occurred.
** May use ferror(), as always, to detect errors.
*/
write(fd, buf, n) unsigned fd, n; unsigned char *buf; {
  unsigned cnt;
  cnt = n;
  while(cnt--) {
    _write(*buf++, fd);
    if(_status[fd] & ERRBIT) return (-1);
    }
  return (n);
  }
```

GETARG.C

```
#include "stdio.h"
/*
** Get command line argument.
** Entry: n    = Number of the argument.
**        s    = Destination string pointer.
**        size = Size of destination string.
**        argc = Argument count from main().
**        argv = Argument vector(s) from main().
** Returns number of characters moved on success,
** else EOF.
*/
getarg(n, s, size, argc, argv)
  int n; char *s; int size, argc, argv[]; {
  char *str;
  int i;
  if(n < 0 | n >= argc) {
    *s = NULL;
    return EOF;
    }
  i = 0;
  str=argv[n];
  while(i<size) {
    if((s[i]=str[i])==NULL) break;
    ++i;
    }
  s[i]=NULL;
  return i;
  }
```

GETCHAR.C

```
#include "stdio.h"
/*
** Get next character from standard input.
*/
getchar() {
  return (fgetc(stdin));
  }
```

IS.C

```
/*
** All character classification functions except isascii().
** Integer argument (c) must be in ASCII range (0-127) for
** dependable answers.
*/

#define ALNUM     1
#define ALPHA     2
#define CNTRL     4
#define DIGIT     8
#define GRAPH    16
#define LOWER    32
#define PRINT    64
#define PUNCT   128
#define BLANK   256
#define UPPER   512
#define XDIGIT 1024


int _is[128] = {
 0x004, 0x004, 0x004, 0x004, 0x004, 0x004, 0x004, 0x004,
 0x004, 0x104, 0x104, 0x104, 0x104, 0x104, 0x004, 0x004,
 0x004, 0x004, 0x004, 0x004, 0x004, 0x004, 0x004, 0x004,
 0x004, 0x004, 0x004, 0x004, 0x004, 0x004, 0x004, 0x004,
 0x140, 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x0D0,
 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x0D0,
 0x459, 0x459, 0x459, 0x459, 0x459, 0x459, 0x459, 0x459,
 0x459, 0x459, 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x0D0,
 0x0D0, 0x653, 0x653, 0x653, 0x653, 0x653, 0x653, 0x253,
 0x253, 0x253, 0x253, 0x253, 0x253, 0x253, 0x253, 0x253,
 0x253, 0x253, 0x253, 0x253, 0x253, 0x253, 0x253, 0x253,
 0x253, 0x253, 0x253, 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x0D0,
 0x0D0, 0x473, 0x473, 0x473, 0x473, 0x473, 0x473, 0x073,
 0x073, 0x073, 0x073, 0x073, 0x073, 0x073, 0x073, 0x073,
 0x073, 0x073, 0x073, 0x073, 0x073, 0x073, 0x073, 0x073,
 0x073, 0x073, 0x073, 0x0D0, 0x0D0, 0x0D0, 0x0D0, 0x004
 };

isalnum (c) int c; {return (_is[c] & ALNUM );} /* 'a'-'z', 'A'-'Z', '0'-
'9' */
isalpha (c) int c; {return (_is[c] & ALPHA );} /* 'a'-'z', 'A'-'Z' */
iscntrl (c) int c; {return (_is[c] & CNTRL );} /* 0-31, 127 */
isdigit (c) int c; {return (_is[c] & DIGIT );} /* '0'-'9' */
```

```
isgraph (c) int c; {return (_is[c] & GRAPH );} /* '!'-' ' */
islower (c) int c; {return (_is[c] & LOWER );} /* 'a'-'z' */
isprint (c) int c; {return (_is[c] & PRINT );} /* ' '-' ' */
ispunct (c) int c; {return (_is[c] & PUNCT );} /* !alnum && !cntrl &&
!space */
isspace (c) int c; {return (_is[c] & BLANK );} /* HT, LF, VT, FF, CR, ' '
*/
isupper (c) int c; {return (_is[c] & UPPER );} /* 'A'-'Z' */
isxdigit(c) int c; {return (_is[c] & XDIGIT);} /* '0'-'9', 'a'-'f', 'A'-
'F' */
```

ISASCII.C

```
/*
** return 'true' if c is an ASCII character (0-127)
*/
isascii(c) unsigned c; {
  return (c < 128);
  }
```

ISATTY.C

```
/*
** Return "true" if fd is a device, else "false"
*/
isatty(fd) int fd; {
fd;                /* fetch handle */
#asm
  push bx          ; save 2nd reg
  mov  bx,ax       ; place handle
  mov  ax,4400h    ; ioctl get info function
  int 21h          ; call BDOS
  pop  bx          ; restore 2nd reg
  mov  ax,dx       ; fetch info bits
  and  ax,80h      ; isdev bit
#endasm
  }
```

ISCONS.C

```
/*
** Determine if fd is the console.
*/
#include <stdio.h>
extern int _cons[];

iscons(fd) int fd; {
  if(_cons[fd] == NULL) {
    if(_iscons(fd)) _cons[fd] = 2;
    else            _cons[fd] = 1;
    }
  if(_cons[fd] == 1) return (NO);
  return (YES);
  }

/*
** Call DOS only the first time for a file.
*/
_iscons(fd) int fd; {
  fd;             /* fetch handle */
#asm
  push bx         ; save 2nd reg
  mov  bx,ax      ; place handle
  mov  ax,4400h   ; ioctl get info function
  int 21h         ; call BDOS
  pop  bx         ; restore 2nd reg
  mov  ax,dx      ; fetch info bits
  and  ax,83h     ; keep device and console bits
  cmp  ax,80h     ; device and console?
  jg   __cons1
  xor  ax,ax      ; return false if not device and console
__cons1:
#endasm
  }
```

ITOA.C

```
/*
** itoa(n,s) - Convert n to characters in s
*/
itoa(n, s) char *s; int n; {
  int sign;
```

```
  char *ptr;
  ptr = s;
  if ((sign = n) < 0) n = -n;
  do {
    *ptr++ = n % 10 + '0';
    } while ((n = n / 10) > 0);
  if (sign < 0) *ptr++ = '-';
  *ptr = '\0';
  reverse(s);
  }
```

ITOAB.C

```
/*
** itoab(n,s,b) - Convert "unsigned" n to characters in s using base b.
**                NOTE: This is a non-standard function.
*/
itoab(n, s, b) int n; char *s; int b; {
  char *ptr;
  int lowbit;
  ptr = s;
  b >>= 1;
  do {
    lowbit = n & 1;
    n = (n >> 1) & 32767;
    *ptr = ((n % b) << 1) + lowbit;
    if(*ptr < 10) *ptr += '0'; else *ptr += 55;
    ++ptr;
    } while(n /= b);
  *ptr = 0;
  reverse (s);
  }
```

ITOD.C

```
#include "stdio.h"
/*
** itod - convert nbr to signed decimal string of width sz
**         right adjusted, blank filled; returns str
**
**        if sz > 0 terminate with null byte
**        if sz = 0 find end of string
**        if sz < 0 use last byte for data
```

```
*/
itod(nbr, str, sz)  int nbr;  char str[];  int sz;  {
  char sgn;
  if(nbr<0) {nbr = -nbr; sgn='-';}
  else sgn=' ';
  if(sz>0) str[-sz]=NULL;
  else if(sz<0) sz = -sz;
  else while(str[sz]!=NULL) ++sz;
  while(sz) {
    str[-sz]=(nbr%10+'0');
    if((nbr=nbr/10)==0) break;
    }
  if(sz) str[-sz]=sgn;
  while(sz>0) str[-sz]=' ';
  return str;
  }
```

ITOO.C

```
/*
** itoo - converts nbr to octal string of length sz
**         right adjusted and blank filled, returns str
**
**        if sz > 0 terminate with null byte
**        if sz = 0 find end of string
**        if sz < 0 use last byte for data
*/
itoo(nbr, str, sz)  int nbr;  char str[];  int sz;  {
  int digit;
  if(sz>0) str[-sz]=0;
  else if(sz<0) sz = -sz;
  else while(str[sz]!=0) ++sz;
  while(sz) {
    digit=nbr&7; nbr=(nbr>>3)&8191;
    str[-sz]=digit+48;
    if(nbr==0) break;
    }
  while(sz) str[-sz]=' ';
  return str;
  }
```

ITOU.C

```
#include "stdio.h"
/*
** itou - convert nbr to unsigned decimal string of width sz
**         right adjusted, blank filled; returns str
**
**        if sz > 0 terminate with null byte
**        if sz = 0 find end of string
**        if sz < 0 use last byte for data
*/
itou(nbr, str, sz)  int nbr;  char str[];  int sz;  {
  int lowbit;
  if(sz>0) str[-sz]=NULL;
  else if(sz<0) sz = -sz;
  else while(str[sz]!=NULL) ++sz;
  while(sz) {
    lowbit=nbr&1;
    nbr=(nbr>>1)&32767;  /* divide by 2 */
    str[-sz]=((nbr%5)<<1)+lowbit+'0';
    if((nbr=nbr/5)==0) break;
    }
  while(sz) str[-sz]=' ';
  return str;
  }
```

ITOX.C

```
/*
** itox - converts nbr to hex string of length sz
**         right adjusted and blank filled, returns str
**
**        if sz > 0 terminate with null byte
**        if sz = 0 find end of string
**        if sz < 0 use last byte for data
*/
itox(nbr, str, sz)  int nbr;  char str[];  int sz;  {
  int digit, offset;
  if(sz>0) str[-sz]=0;
  else if(sz<0) sz = -sz;
  else while(str[sz]!=0) ++sz;
  while(sz) {
    digit=nbr&15; nbr=(nbr>>4)&4095;
    if(digit<10) offset=48; else offset=55;
```

```
    str[-sz]=digit+offset;
    if(nbr==0) break;
    }
  while(sz) str[-sz]=' ';
  return str;
  }
```

LEFT.C

```
/*
** left - left adjust and null terminate a string
*/
left(str) char *str; {
  char *str2;
  str2=str;
  while(*str2==' ') ++str2;
  while(*str++ = *str2++);
  }
```

LEXCMP.C

```
char _lex[128] = {
       0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  /* NUL - /       */
      10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
      20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
      30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
      40, 41, 42, 43, 44, 45, 46, 47,
      65, 66, 67, 68, 69, 70, 71, 72, 73, 74,  /* 0-9           */
      48, 49, 50, 51, 52, 53, 54,              /* : ; < = > ? @ */
      75, 76, 77, 78, 79, 80, 81, 82, 83, 84,  /* A-Z           */
      85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
      95, 96, 97, 98, 99,100,
      55, 56, 57, 58, 59, 60,                  /* [ \ ] ^ _ `   */
      75, 76, 77, 78, 79, 80, 81, 82, 83, 84,  /* a-z           */
      85, 86, 87, 88, 89, 90, 91, 92, 93, 94,
      95, 96, 97, 98, 99,100,
      61, 62, 63, 64,                          /* { | }         */
     127};                                     /* DEL           */

/*
** lexcmp(s, t) - Return a number <0, 0, or >0
**                as s is <, =, or > t.
```

```
*/
lexcmp(s, t) char *s, *t; {
  while(lexorder(*s, *t) == 0)
    if(*s++) ++t;
    else return (0);
  return (lexorder(*s, *t));
  }

/*
** lexorder(c1, c2)
**
** Return a negative, zero, or positive number if
** c1 is less than, equal to, or greater than c2,
** based on a lexicographical (dictionary order)
** colating sequence.
**
*/
lexorder(c1, c2) int c1, c2; {
  return(_lex[c1] - _lex[c2]);
  }
```

MALLOC.C

```
#include "stdio.h"
/*
** Memory allocation of size bytes.
** size  = Size of the block in bytes.
** Returns the address of the allocated block,
** else NULL for failure.
*/
malloc(size) unsigned size; {
  return (_alloc(size, NO));
  }
```

OTOI.C

```
#include "stdio.h"
/*
** otoi - convert unsigned octal string to integer nbr
**          returns field size, else ERR on error
*/
otoi(octstr, nbr)  char *octstr;  int *nbr;  {
  int d,t; d=0;
```

```
  *nbr=0;
  while((*octstr>='0')&(*octstr<='7')) {
    t=*nbr;
    t=(t<<3) + (*octstr++ - '0');
    if ((t>=0)&(*nbr<0)) return ERR;
    d++; *nbr=t;
    }
  return d;
  }
```

PAD.C

```
/*
** Place n occurrences of ch at dest.
*/
pad(dest, ch, n) char *dest; unsigned n, ch; {
  while(n--) *dest++ = ch;
  }
```

POLL.C

```
#include "stdio.h"
#include "clib.h"
/*
** Poll for console input or interruption
*/
poll(pause) int pause; {
  int i;
  if(i = _hitkey())  i = _getkey();
  if(pause) {
    if(i == PAUSE) {
      i = _getkey();            /* wait for next character */
      if(i == ABORT) exit(2);   /* indicate abnormal exit */
      return (0);
      }
    if(i == ABORT) exit(2);
    }
  return (i);
  }
```

PUTCHAR.C

```
#include "stdio.h"
/*
** Write character to standard output.
*/
putchar(ch) int ch; {
  return (fputc(ch, stdout));
  }
```

PUTS.C

```
#include "stdio.h"
/*
** Write string to standard output.
*/
puts(string) char *string; {
  fputs(string, stdout);
  fputc('\n', stdout);
  }
```

RENAME.C

```
#include "stdio.h"
#include "clib.h"
/*
** Rename a file.
**  from = address of old filename.
**    to = address of new filename.
**  Returns NULL on success, else ERR.
*/
rename(from, to) char *from, *to; {
  if(_rename(from, to)) return (NULL);
  return (ERR);
  }

_rename(old, new) char *old, *new; {
#asm
  push ds          ; ds:dx points to old name
  pop  es          ; es:di points to new name
  mov  di,[bp+4]   ; get "new" offset
  mov  dx,[bp+6]   ; get "old" offset
  mov  ah,56h      ; rename function
  int  21h         ; call bdos
```

```
  jnc  __ren1      ; error?
  xor  ax,ax       ; yes, return false
  jmp  __ren2
__ren1:            ; no, set hi and lo
  mov  ax,1        ; return true
__ren2:
#endasm
  }
```

REVERSE.C

```
/*
** reverse string in place
*/
reverse(s) char *s; {
  char *j;
  int c;
  j = s + strlen(s) - 1;
  while(s < j) {
    c = *s;
    *s++ = *j;
    *j- = c;
    }
  }
```

REWIND.C

```
/*
** Rewind file to beginning.
*/
rewind(fd) int fd; {
  return(cseek(fd, 0, 0));
  }
```

SIGN.C

```
/*
** sign - return -1, 0, +1 depending on the sign of nbr
*/
sign(nbr)  int nbr;  {
  if(nbr > 0)  return 1;
  if(nbr == 0) return 0;
```

```
  return -1;
  }
```

STRCAT.C

```
/*
** concatenate t to end of s
** s must be large enough
*/
strcat(s, t) char *s, *t; {
  char *d;
  d = s;
  -s;
  while (*++s) ;
  while (*s++ = *t++) ;
  return (d);
  }
```

STRCHR.C

```
/*
** return pointer to 1st occurrence of c in str, else 0
*/
strchr(str, c) char *str, c; {
  while(*str) {
    if(*str == c) return (str);
    ++str;
    }
  return (0);
  }
```

STRCMP.C

```
/*
** return <0,   0,  >0 a_ording to
**       s<t, s=t, s>t
*/
strcmp(s, t) char *s, *t; {
  while(*s == *t) {
    if(*s == 0) return (0);
    ++s; ++t;
    }
```

```
  return (*s - *t);
  }
```

STRCPY.C

```
/*
** copy t to s
*/
strcpy(s, t) char *s, *t; {
  char *d;
  d = s;
  while (*s++ = *t++) ;
  return (d);
  }
```

STRLEN.C

```
/*
** return length of string s (fast version)
*/
strlen(s) char *s; {
  #asm
  xor al,al         ; set search value to zero
  mov cx,65535      ; set huge maximum
  mov di,[bp+4]     ; get address of s
  cld               ; set direction flag forward
  repne scasb       ; scan for zero
  mov ax,65534
  sub ax,cx         ; calc and return length
  #endasm
  }
```

STRNCAT.C

```
/*
** concatenate n bytes max from t to end of s
** s must be large enough
*/
strncat(s, t, n) char *s, *t; int n; {
  char *d;
  d = s;
  -s;
```

```
  while(*++s) ;
  while(n-) {
    if(*s++ = *t++) continue;
    return(d);
    }
  *s = 0;
  return(d);
  }
```

STRNCMP.C

```
/*
** strncmp(s,t,n) - Compares two strings for at most n
**                  characters and returns an integer
**                  >0, =0, or <0 as s is >t, =t, or <t.
*/
strncmp(s, t, n) char *s, *t; int n; {
  while(n && *s==*t) {
    if (*s == 0) return (0);
    ++s; ++t; -n;
    }
  if(n) return (*s - *t);
  return (0);
  }
```

STRNCPY.C

```
/*
** copy n characters from sour to dest (null padding)
*/
strncpy(dest, sour, n) char *dest, *sour; int n; {
  char *d;
  d = dest;
  while(n- > 0) {
    if(*d++ = *sour++) continue;
    while(n- > 0) *d++ = 0;
    }
  *d = 0;
  return (dest);
  }
```

STRRCHR.C

```
/*
** strrchr(s,c) - Search s for rightmost occurrance of c.
** s      = Pointer to string to be searched.
** c      = Character to search for.
** Returns pointer to rightmost c or NULL.
*/
strrchr(s, c) char *s, c; {
  char *ptr;
  ptr = 0;
  while(*s) {
    if(*s==c) ptr = s;
    ++s;
    }
  return (ptr);
  }
```

TOASCII.C

```
/*
** return ASCII equivalent of c
*/
toascii(c) int c; {
  return (c);
  }
```

TOLOWER.C

```
/*
** return lower-case of c if upper-case, else c
*/
tolower(c) int c; {
  if(c<='Z' && c>='A') return (c+32);
  return (c);
  }
```

TOUPPER.C

```
/*
** return upper-case of c if it is lower-case, else c
*/
toupper(c) int c; {
```

```
  if(c<='z' && c>='a') return (c-32);
  return (c);
  }
```

UNGETC.C

```
#include "stdio.h"
extern _nextc[];
/*
** Put c back into file fd.
** Entry:  c = character to put back
**         fd = file descriptor
** Returns c if successful, else EOF.
*/
ungetc(c, fd) int c, fd; {
  if(!_mode(fd) || _nextc[fd]!=EOF || c==EOF) return (EOF);
  return (_nextc[fd] = c);
  }
```

UNLINK.C

```
#include "stdio.h"
#include "clib.h"
/*
** Unlink (delete) the named file.
** Entry: fn = Null-terminated DOS file path\name.
** Returns NULL on success, else ERR.
*/
unlink(fn) char *fn; {
  fn;           /* load fn into ax */
#asm
  mov dx,ax     ; put fn in its place
  mov ah,41h    ; delete function code
  int 21h
  mov ax,0
  jnc __unlk    ; return NULL
  mov ax,-2     ; return ERR
__unlk:
#endasm
  }
#asm
_delete: jmp    _unlink
         public _delete
```

```
#endasm
```

UTOI.C

```
#include "stdio.h"
/*
** utoi - convert unsigned decimal string to integer nbr
**          returns field size, else ERR on error
*/
utoi(decstr, nbr)  char *decstr;  int *nbr;  {
  int d,t; d=0;
  *nbr=0;
  while((*decstr>='0')&(*decstr<='9')) {
    t=*nbr;t=(10*t) + (*decstr++ - '0');
    if ((t>=0)&(*nbr<0)) return ERR;
    d++; *nbr=t;
    }
  return d;
  }
```

XTOI.C

```
#include stdio.h
/*
** xtoi - convert hex string to integer nbr
**         returns field size, else ERR on error
*/
xtoi(hexstr, nbr) char *hexstr; int *nbr; {
  int d, b;  char *cp;
  d = *nbr = 0; cp = hexstr;
  while(*cp == '0') ++cp;
  while(1) {
    switch(*cp) {
      case '0': case '1': case '2':
      case '3': case '4': case '5':
      case '6': case '7': case '8':
      case '9':                      b=48; break;
      case 'A': case 'B': case 'C':
      case 'D': case 'E': case 'F': b=55; break;
      case 'a': case 'b': case 'c':
      case 'd': case 'e': case 'f': b=87; break;
       default: return (cp - hexstr);
      }
```

```
  if(d < 4) ++d; else return (ERR);
  *nbr = (*nbr << 4) + (*cp++ - b);
  }
}
```

APPENDIX E

Small C Quick Reference Guide

This guide is not a formal definition of the Small C language. Rather, it is intended to provide quickly accessible information about the syntax of the language. Generic terms are written as one or more words in italics with the leading letter of each word capitalized. Keywords and special characters in boldface are required by the syntax. The word ***String*** implies a sequence of characters written together. The word ***List*** implies a series of the preceding item separated by commas and optional white space. The ellipsis (...) implies the optional repetition of occurrences of the preceding type of item. A question mark at the end of an item means that the item is optional, and may be omitted.

LANGUAGE SYNTAX

ArgumentDeclaration:

ObjectDeclaration

ArgumentList:

NameList

Directive:

#include "*Filename*"
#include <*Filename*>
#include *Filename*
#define *Name CharacterString?...*
#ifdef *Name*
#ifndef *Name*
#else
#endif
#asm
#endasm

Constant:

```
        Integer
        'Character'             (escape sequence allowed)
        'CharacterCharacter'    (escape sequences allowed)
```

ConstantExpression:

```
        Constant
        Operator ConstantExpression
        ConstantExpression Operator ConstantExpression
        (ConstantExpression)
```

Declarator:

```
        Object Initializer?     (global initializers only)
```

EscapeSequence:

```
        \n              (newline)
        \t              (tab)
        \b              (backspace)
        \f              (formfeed)
        \OctalInteger
        \OtherCharacter
```

Expression:

```
        Primary
        Operator Expression
        Expression Operator
        Expression Operator Expression
```

FunctionDeclaration:

```
        void? Name (ArgumentList?)
                ArgumentDeclaration?...
                CompoundStatement
```

GlobalDeclaration:

ObjectDeclaration
FunctionDeclaration

Initializer:

= *ConstantExpression*
= {*ConstantExpressionList*}
= *StringConstant*

Object:

Name
**Name*
Name [*ConstantExpression?*]
Name()
(**Name*)() (arguments and locals only)

ObjectDeclaration:

Type DeclaratorList;
extern *Type? DeclaratorList*; (global only)

Primary:

Name
Constant
StringConstant
Name [*Expression*]
Primary (*ExpressionList?*)
(*Expression*)

Program:

Directive?... GlobalDeclaration...

Statement:

```
;
ExpressionList;
return ExpressionList?;
Name:
goto Name;
if (ExpressionList) Statement
if (ExpressionList) Statement else Statement
switch (ExpressionList) CompoundStatement
case ConstantExpression:
default:
break;
while (ExpressionList) Statement
for(ExpressionList?;
    ExpressionList?;
    ExpressionList?) Statement
do Statement while (ExpressionList);
continue;
{ObjectDeclaration?... Statement?...}
```

StringConstant:

```
"CharacterString"     (escape sequences allowed)
```

Type:

```
char
int
unsigned
unsigned char
unsigned int
```

STANDARD FUNCTIONS

Function	Returns
abs (nbr) int nbr;	absolute value
abort (errcode) int errcode;	
atoi (str) char *str;	value
atoib (str, base) char *str; int base;	value
auxbuf(fd, sz) int fd; char *sz;	zero, ERR
avail (abort) int abort;	# bytes available, zero
bseek(fd, offset, from) int fd, offset[], from;	zero, EOF
btell(fd, offset) int fd, offset[];	byte number -> offset
calloc (nbr, sz) int nbr, sz;	pointer, zero
cfree (addr) char *addr;	addr, zero
clearerr (fd) int fd;	
cseek (fd, offset, from) int fd, offset, from;	zero, EOF
ctell (fd) int fd;	block number
ctellc(fd) int fd;	byte number in block
delete (name) char *name;	zero, ERR
dtoi (str, nbr) char *str; int *nbr;	field length
exit (errcode) int errcode;	
fclose (fd) int fd;	zero, non-zero
feof (fd) int fd;	non-zero, zero
ferror (fd) int fd;	non-zero, zero
fgetc (fd) int fd;	character, EOF

Function	Returns
`fgets (str, sz, fd) char *str; int sz, fd;`	str, zero
`fopen (name, mode) char *name, *mode;`	fd, zero
`fprintf (fd, str, arg1, arg2, ...)` `int fd; char *str;`	# characters printed
`fputc (c, fd) char c; int fd;`	c, EOF
`fputs (str, fd) char *str; int fd;`	
`fread (ptr, sz, cnt, fd)` `char *ptr; int sz, cnt, fd;`	# items read
`free (addr) char *addr;`	addr, zero
`freopen (name, mode, fd)` `char *name, *mode; int fd;`	fd, zero
`fscanf (fd, str, arg1, arg2, ...)` `int fd; char *str;`	# fields scanned, EOF
`fwrite (ptr, sz, cnt, fd)` `char *ptr; int sz, cnt, fd;`	# items written
`getarg (nbr, str, sz, argc, argv)` `char *str; int nbr, sz, argc, *argv;`	field length, EOF
`getc (fd) int fd;`	character, EOF
`getchar ()`	character, EOF
`gets (str) char *str;`	str, zero
`isalnum (c) char c;`	non-zero, zero
`isalpha (c) char c;`	non-zero, zero
`isascii (c) char c;`	non-zero, zero
`isatty (fd) int fd;`	non-zero, zero

Function	Returns
iscntrl (c) char c;	non-zero, zero
iscons (fd) int fd;	non-zero, zero
isdigit (c) char c;	non-zero, zero
isgraph (c) char c;	non-zero, zero
islower (c) char c;	non-zero, zero
isprint (c) char c;	non-zero, zero
ispunct (c) char c;	non-zero, zero
isspace (c) char c;	non-zero, zero
isupper (c) char c;	non-zero, zero
isxdigit (c) char c;	non-zero, zero
itoa (nbr, str) int nbr; char *str;	
itoab (nbr, str, base) int nbr; char *str; int base;	
itod (nbr, str, sz) int nbr, sz; char *str;	str
itoo (nbr, str, sz) int nbr, sz; char *str;	str
itou (nbr, str, sz) int nbr, sz; char *str;	str
itox (nbr, str, sz) int nbr, sz; char *str;	str
left (str) char *str;	
lexcmp (str1, str2) char *str1, *str2;	<0, 0, >0
lexorder (c1, c2) char c1, c2;	<0, 0, >0
malloc (nbr) int nbr;	pointer, zero
otoi (str, nbr) char *str; int *nbr;	field length
pad(dest, ch, n) char *dest, ch; int n;	

Function	Returns
`poll (pause) int pause;`	character, zero
`printf (str, arg1, arg2, ...) char *str;`	# characters printed
`putc (c, fd) char c; int fd;`	c, EOF
`putchar (c) char c;`	c, EOF
`puts (str) char *str;`	
`read (fd, ptr, cnt) int fd, cnt; char *ptr;`	# bytes read
`reverse (str) char *str;`	
`rewind (fd) int fd;`	zero, EOF
`scanf (str, arg1, arg2, ...) char *str;`	# fields scanned, EOF
`sign (nbr) int nbr;`	-1, 0, +1
`strcat (dest, sour) char *dest, *sour;`	dest
`strchr (str, c) char *str, c;`	pointer, zero
`strcmp (str1, str2) char *str1, *str2;`	<0, 0, >0
`strcpy (dest, sour) char *dest, *sour;`	dest
`strlen (str) char *str;`	string length
`strncat (dest, sour, n) char *dest, *sour; int n;`	dest
`strncmp (str1, str2, n) char *str1, *str2; int n;`	<0, 0, >0
`strncpy (dest, sour, n) char *dest, *sour; int n;`	dest
`strrchr (str, c) char *str, c;`	pointer, zero
`toascii (c) char c;`	ASCII value

Function	Returns
tolower (c) char c;	lowercase
toupper (c) char c;	uppercase
ungetc (c, fd) char c; int fd;	c, EOF
unlink (name) char *name;	zero, ERR
utoi (str, nbr) char *str; int *nbr;	field length
write (fd, ptr, cnt) int fd, cnt; char *ptr;	# bytes written
xtoi (str, nbr) char *str; int *nbr;	field length

APPENDIX F

Small C Error Messages

When the compiler encounters an error condition it displays the erroneous line, an arrow pointing up to the approximate location of the error, and an explanatory error message. If instructed by the **-P** switch, the compiler pauses, waiting for the operator to press the **ENTER** key.

Some programs may cause the compiler to overflow one of its internal buffers. If that happens two alternatives are available: (1) recompile the compiler with more space for the offending buffer, or (2) revise the program to avoid the overflow condition.

The error messages which the compiler generates are listed below in alphabetic order with explanations. The explanations describe the intended use of the messages. Sometimes, however, an error message will surface because of conditions other than those anticipated by the compiler. So the text of a message may not always be entirely appropriate to the circumstances.

```
already defined
```

A name is being declared more than once at the global level or among the formal arguments of a function.

```
bad label
```

A **goto** statement has an improperly formed label. Either it does not conform to the C naming conventions or it is missing altogether.

```
can't subscript
```

A subscript is associated with something which is neither a pointer nor an array.

```
cannot assign to pointer
```

An initializer consisting of a constant or a constant expression is associated with a pointer. Integer pointers do not take initializers and character pointers take only expression-list or string initializers.

```
control statement nesting limit
```

The level of nesting of any combination of **do**, **for**, **while**, and **switch** statements exceeds the capacity of the while queue. This may be corrected by increasing the size of **WQTABSZ** in **CC.H**. **WQTABSZ** is the size (in bytes) of the while queue. It must be a multiple of **WQSIZ**.

```
global symbol table overflow
```

The global symbol table has overflowed. This may be corrected by recompiling the compiler with larger values assigned to the symbols **NUMGLBS** and **SYMTBSZ** in **CC.H**. **NUMGLBS** is the number of global entries the table will hold, and **SYMTBSZ** is the overall size (in bytes) of the combined local and global tables. A comment in the source text explains how to calculate **SYMTBSZ**.

```
illegal address
```

The address operator is applied to something which is neither a variable, a pointer, nor a subscripted pointer or array name.

```
illegal argument name
```

A name in a function's formal argument list does not conform to the C naming conventions.

```
illegal function or declaration
```

The compiler found something at the global level which is not a function or object declaration.

`illegal symbol`

The compiler found a symbol which does not conform to the C naming convention.

`invalid expression`

An expression contains a primary which is neither a constant, a string, nor a valid C name. Note that previously undeclared names (if they are correctly formed) are assumed to be function names and do not produce this error.

`line too long`

A source line, after preprocessing, is more than **LINEMAX** characters long. This can be corrected by breaking the line into parts. However, the compiler can be recompiled with larger values for **LINEMAX** and **LINESIZE** in **CC.H**. Note that **LINESIZE** must be one greater than **LINEMAX**.

`literal queue overflow`

A string constant would overflow the compiler's literal pool. The literal pool may be increased by recompiling the compiler with a larger value for **LITABSZ** in **CC.H**. **LITABSZ** is the size of the literal buffer in bytes.

`local symbol table overflow`

A local declaration would overflow the local symbol table. This may be corrected by recompiling the compiler with larger values for **NUMLOCS** and **SYMTBSZ** in **CC.H**. **NUMLOCS** is the number of entries in the table, and **SYMTBSZ** is the overall size (in bytes) of the combined local and global symbol tables. A comment in the source text shows how to calculate **SYMTBSZ**.

`macro name table full`

A **#define** command would overflow the macro name table. The table may be expanded by recompiling the compiler with larger values for **MACNBR** in **CC.H**. **MACNBR** is the number of names that will fit into the table.

```
macro string queue full
```

A **#define** command would overflow the macro string queue. This may be resolved by recompiling the compiler with a larger value for **MACQSIZE** in **CC.H**. **MACQSIZE** is the size of the macro-string buffer in bytes and is calculated as **MACNBR*7**. You could increase **MACNBR** or assume more than 7 bytes per substitution string on the average. The macro-string buffer must hold all of the replacement strings defined for the entire program being compiled.

```
mismatched expressions
```

The second and third expressions of a conditional operator (*expr1* **?** *expr2* **:** *expr3*) are not compatible. The compiler does not know to determine the attributes of the result.

```
missing token
```

The syntax requires a particular token which is missing.

```
multiple defaults
```

A **switch** contains more than one **default** prefix.

```
must assign to char pointer or char array
```

A string initializer is applied to something other than a character pointer or a character array.

```
must be constant expression
```

Something other than a constant expression was found where the syntax requires a constant expression.

```
must be lvalue
```

Something other than an lvalue is used as a receiving field in an expression. An **lvalue** is an expression (possibly just a name) for a storage location in memory which may be altered. Assigning something to a constant or an unsubscripted array name will produce this message.

```
must be object or type
```

The **sizeof** operator refers to something besides an object name or a type specification.

```
must declare first in block
```

A local declaration occurs after the first executable statement in a block.

```
need array size
```

A local array declaration does not specify the number of elements in the array.

```
negative size illegal
```

An array is dimensioned to a negative value. Recall that constant expressions may be used as array dimensions. Such an expression may very well evaluate to a negative value.

```
no apostrophe
```

A character constant lacks its terminating apostrophe.

```
no comma
```

An argument or declaration list lacks a separating comma.

```
no final }
```

The end of the input occurred while inside of a compound statement.

```
no matching #if...
```

An **#else** or **#endif** is not preceded by a corresponding **#ifdef** or **#ifndef** directive.

```
no open paren
```

An apparent function declarator lacks the left parenthesis which introduces the formal argument list.

```
no quote
```

A string constant lacks its terminating double quote. This quotation mark must be on the same line as the initial quotation mark.

```
no semicolon
```

A semicolon does not appear where the syntax requires one.

```
not a label
```

The name following the keyword **goto** is defined, but not as a label.

```
not allowed in switch
```

A local declaration occurs within the body of a **switch** statement. Small C disallows that possibility.

```
not allowed with block-locals
```

A **goto** statement occurs in a function which has local declarations in blocks below the highest level. Small C disallows that situation.

```
not allowed with goto
```

A local declaration occurs in a block below the highest level in a function which contains **goto** statements. Small C disallows that possibility.

```
not an argument
```

The names in a function's formal argument list do not match the corresponding type declarations.

```
not in switch
```

A **case** or **default** prefix occurs outside of a **switch** statement.

```
open failure on include file
```

An **#include** file cannot be opened.

`out of context`

A **break** statement is not located within a **do**, **for**, **while**, or **switch** statement, or a **continue** is not within a **do**, **for**, or **while** statement.

`staging buffer overflow`

The code generated by an expression exceeds the size of the staging buffer. This can be corrected by breaking the expression into several intermediate expressions, or by recompiling the compiler with a larger **STAGESIZE** in **CC.H**. **STAGESIZE** is the number of 4-byte entries in the staging buffer.

`too many cases`

The number of **case** prefixes in a **switch** exceeds the capacity of the switch table. The switch table may be enlarged by assigning a larger value to **SWTABSZ** in **CC.H** and recompiling the compiler. **SWTABSZ** is the size (in bytes) of the switch table. It must be a multiple of **SWSIZ**.

`try (*...)()`

An argument or local declaration specifies a function, instead of a function pointer.

`wrong number of arguments`

One or more of the formal arguments in a function header was not typed before entering the function body.

APPENDIX G

Changes from Small C 2.1

The most popular MS-DOS version of the Small C compiler to date is version 2.1 as distributed by M&T Books. With the publication of this book, however, a new version (2.2) was released. The following material is provided to help you assess the differences and thereby determine how applicable this book may be to the compiler you may already have and whether or not to obtain a copy of the new version.

1. Several bugs have been fixed.

a. The mapping of special keystrokes obtained through the various **get** and **read** functions was not consistent with **poll().**

b. **Scanf()** and **fscanf()** dropped the first character of an input field because **fgetc()** didn't look for ungotten characters.

c. **Ispunct()** failed to exclude white space from the *true* condition.

d. **Unlink()** returned **ERR** on failure instead of **EOF.**

e. The expression analyzer lost pointer offsets when subscripting was applied; for example, **(ptr+5)[x]** would lose the offset 5.

f. **Hier2()** (now called **level2()),** which processes the **?:** operator, didn't group properly or yield meaningful attributes for further expression analysis.

2. The optimizer has been entirely rewritten for greater efficiency, maintainability, and understandability. It now does much more optimizing. EXE sizes are down by 10% typically and execution times are down by as much as 25%.

3. **Iscons()** was revised to call DOS only the first time for a file and remember the answer thereafter. The effect was a DOUBLING OF I/O SPEED through the **get** functions.

4. **Strlen()** was revised to use the 8086 **REPNE SCASB** prefix/instruction for the fastest possible operation. This sped up finding a string's length by over five times.

5. The **is...()** character classification functions have been consolidated into a single module with an array of 128 bytes which are encoded with the various attributes of the ASCII characters. These functions now simply subscript the array and select the appropriate bit. This approach is three times faster than before.

6. Previously optional compiler features have been made standard. Primarily this includes dynamic allocation of data structures and support for flow-of-control statements that were not in the original Small C compiler.

7. A general clean up of the source code was performed. This involved consolidating the compiler code into just four files, reorganizing the placement of functions, renaming the p-codes in a systematic way, renaming many functions, purging unused variables, reducing the number of functions, and numerous minor code improvements.

8. The amount of stack space used by the expression analyzer was reduced by allocating instances of **lval2[]** (now called **is2[]**) only when actually needed.

9. Support for the **unsigned** integer and character data types was added.

10. The handling of numeric constants was revised to recognize values from 32768 to 65535 as unsigned, and treat them as such during expression analysis.

11. Rather than defaulting declarations with multiple modifiers (for example, ***array**[]) to something the programmer doesn't intend, they are now rejected.

12. The compiler now accepts **void** before function headers as a comment.

13. The implementation of the **sizeof** operator was revised to work correctly. Although undocumented, this operator was in the original MS-DOS compiler, but it was incorrectly implemented.

14. An argument count is now always passed to called functions—even if **NOCCARGC** is defined. This is not very costly on 80x86 family CPUs and it eliminates a lot of headaches when you call **printf()** but forget that **NOCCARGC** is defined.

15. Small C programs now abort with a return code of 1 for memory overflows and 2 for operator interruptions (<control-C>).

APPENDIX H

Index of Compiler Functions

Function	CCx	Txt	Lst
addlabel	1	256	452
addsym	2	256	461
addwhile	2	257	462
alpha	2	284	463
amatch	2	286	459
an	2	285	463
ask	1	292	453
astreq	2	285	459
blanks	2	284	460
bump	2	284	460
calc	3	352	479
calc2	3		479
callfunc	3	367	472
chrcon	3	374	475
clearstage	4	264	488
colon	4	266	493
compound	1	313	448
constant	3	371	474
constexpr	3	335	465
decl	1	312	447
declglb	1	297	442
declloc	1	310	448
delwhile	2	257	463
doargs	1	303	446
doasm	1	301	452
dobreak	1	323	452
docase	1	322	451
docont	1	323	452
dodeclare	1	297	442
dodefault	1	323	451
dodefine	1	301	444
dodo	1	318	449
doexpr	1	335	453
dofor	1	319	450
dofunction	1	302	445
dogoto	1	324	451
doif	1	315	449
doinclude	1	301	444
dolabel	1	325	451
doreturn	1	325	452
doswitch	1	320	450
double	3		473
dowhile	1	317	449
down	3	343	477
down1	3	345	477
down2	3	346	477
dropout	3	343	476
dumplits	4	265	490
dumpstage	4	265	489
dumpzero	4	266	491
endst	2		461
error	2		464
errout	2		464
experr	3		472
expression	3	336	465
external	4	262	490
fetch	3		473
findglb	2	256	462
findloc	2	256	462
gch	2	284	460
gen	4	263	488
getint	2		463
getlabel	2		463
getpop	4	392	492
hash	2		462
header	4	259	487
ifline	2	288	457
illname	2		464
inbyte	2	283	458
init	1	300	443
initials	1	299	443
inline	2	290	457
isfree	4	392	492
keepch	2	283	457
kill	2		460
level1	3	353	466
level2	3	355	467
level3	3	357	468
level4	3	357	468
level5	3	357	468
level6	3	357	468
level7	3	357	468
level8	3	357	468
level9	3	357	468
level10	3	357	468
level11	3	357	468
level12	3	357	468
level13	3	357	468
level14	3	363	470
litchar	3	375	475
lout	2		464
main	1	291	441
match	2	286	459
multidef	2		464
mustopen	1		454
need	2		458
needlval	2		464
needsub	1		444
newline	4	267	493
nextop	2	286	459
nextsym	2		462
noiferr	2		464
nosign	3		478
ns	2		458
number	3	373	474
openfile	1	290	454
outcode	4	272	493
outdec	4	267	494
outline	4	267	494
outname	4	267	495
outsize	4	262	490
outstr	4	267	495
parse	1	296	442
peep	4	390	491
point	4	262	490
preprocess	2	287	455
primary	3	368	471
public	4	261	489
putint	2		464
putmac	1		445
readwhile	2	257	463
search	2	256	462
setcodes	4	293	484
setseq	4	293	484
setstage	4	262	487
skim	3	341	476
skip	2		461
statement	1	308	447
step	3	360	473
store	3	353	473
stowlit	3	375	475
streq	2	285	459
string	3	374	475
symname	2	285	458
test	3	338	465
toseg	4	259	489
trailer	4	260	487
white	2	284	460
zerojump	3	340	466

APPENDIX I

Index of Library Functions

Function	Module	Lst	Function	Module	Lst	Function	Module	Lst
_adjust	CSYSLIB.C	510	ccargc	CALL.ASM	502	isxdigit	IS.C	526
_alloc	CSYSLIB.C	511	cfree	FREE.C	521	itoa	ITOA.C	527
_backup	CSYSLIB.C	510	clearerr	CLEARERR.C	504	itoab	ITOAB.C	528
_bdos2	CSYSLIB.C	511	cseek	CSEEK.C	504	itod	ITOD.C	528
_clreof	CSYSLIB.C	511	cseekc	CSEEK.C	505	itoo	ITOO.C	529
_clrerr	CSYSLIB.C	511	ctell	CTELL.C	513	itou	ITOU.C	529
_empty	CSYSLIB.C	510	ctellc	CTELL.C	513	itox	ITOX.C	530
_eq	CALL.ASM	502	delete	UNLINK.C	538	left	LEFT.C	530
_flush	CSYSLIB.C	509	dtoi	DTOI.C	514	lexcmp	LEXCMP.C	531
_ge	CALL.ASM	502	exit	EXIT.C	514	lexorder	LEXCMP.C	531
_getkey	CSYSLIB.C	513	fclose	FCLOSE.C	515	malloc	MALLOC.C	531
_gt	CALL.ASM	502	feof	FEOF.C	515	otoi	OTOI.C	532
_hitkey	CSYSLIB.C	512	ferror	FERROR.C	515	pad	PAD.C	532
_le	CALL.ASM	502	fgetc	FGETC.C	515	poll	POLL.C	532
_lneg	CALL.ASM	503	fgets	FGETS.C	516	printf	FPRINTF.C	518
_lt	CALL.ASM	502	fopen	FOPEN.C	518	putc	FPUTC.C	520
_main	CSYSLIB.C	506	fprintf	FPRINTF.C	518	putchar	PUTCHAR.C	532
_mode	CSYSLIB.C	510	fputc	FPUTC.C	519	puts	PUTS.C	533
_ne	CALL.ASM	502	fputs	FPUTS.C	520	read	FREAD.C	520
_open	CSYSLIB.C	507	fread	FREAD.C	520	rename	RENAME.C	533
_parse	CSYSLIB.C	506	free	FREE.C	521	reverse	REVERSE.C	534
_read	CSYSLIB.C	508	freopen	FREOPEN.C	521	rewind	REWIND.C	534
_readbuf	CSYSLIB.C	508	fscanf	FSCANF.C	522	scanf	FSCANF.C	522
_seek	CSYSLIB.C	512	fwrite	FWRITE.C	524	sign	SIGN.C	534
_seteof	CSYSLIB.C	511	getarg	GETARG.C	524	strcat	STRCAT.C	534
_seterr	CSYSLIB.C	511	getc	FGETC.C	516	strchr	STRCHR.C	535
_switch	CALL.ASM	503	getchar	GETCHAR.C	525	strcmp	STRCMP.C	535
_uge	CALL.ASM	502	gets	FGETS.C	517	strcpy	STRCPY.C	535
_ugt	CALL.ASM	502	gets	FGETS.C	517	strlen	STRLEN.C	535
_ule	CALL.ASM	502	isalnum	IS.C	526	strncat	STRNCAT.C	536
_ult	CALL.ASM	502	isalpha	IS.C	526	strncmp	STRNCMP.C	536
_write	CSYSLIB.C	509	isascii	ISASCII.C	526	strncpy	STRNCPY.C	536
_writebuf	CSYSLIB.C	509	isatty	ISATTY.C	526	strrchr	STRRCHR.C	537
abort	EXIT.C	514	iscntrl	IS.C	526	toascii	TOASCII.C	537
abs	ABS.C	498	iscons	ISCONS.C	527	tolower	TOLOWER.C	537
atoi	ATOI.C	498	isdigit	IS.C	526	toupper	TOUPPER.C	537
atoib	ATOIB.C	499	isgraph	IS.C	526	ungetc	UNGETC.C	538
auxbuf	AUXBUF.C	499	islower	IS.C	526	unlink	UNLINK.C	538
avail	AVAIL.C	500	isprint	IS.C	526	utoi	UTOI.C	538
bseek	BSEEK.C	500	ispunct	IS.C	526	write	FWRITE.C	524
btell	BTELL.C	501	isspace	IS.C	526	xtoi	XTOI.C	539
calloc	CALLOC.C	504	isupper	IS.C	526			

Bibliography

History of Small C

[1] Cain, Ron. "A Small C Compiler for the 8080's." *Dr. Dobb's Journal*, April-May 1980, pp. 5-19.

[2] — "A Runtime Library for the Small C Compiler." *Dr. Dobb's Journal*, September 1980, pp. 4-15.

[3] Hendrix, J. E. "Small-C Expression Analyzer." *Dr. Dobb's Journal*, December 1981, pp. 40-43.

[4] Hendrix, J. E. "Small-C Compiler, v.2." *Dr. Dobb's Journal*, December 1982, pp. 16-53, and January 1983, pp. 48-64.

[5] Hendrix, J. E., and L. E. Payne. "A New Library for Small-C." *Dr. Dobb's Journal*, May 1984, pp. 50-81, and June 1984, pp. 56-69.

[6] Hendrix, J. E. "Small-C Update." *Dr. Dobb's Journal*, August 1985, pp. 84-91.

Assembly Language Programming

[7] Morgan, Christopher L. *Bluebook of Assembly Routines for the IBM PC & XT*. New York: The Plume/Waite Group, 1984.

[8] Morgan, Christopher L., and Mitchell Waite. *8086/8088 16-Bit Microprocessor Primer*. Peterborough, N.J.: BYTE/McGraw-Hill, 1982.

C Language Programming

[9] Johnson, S. C., B. W. Kernighan, M. E. Lesk, and D. M. Ritchie. "The C Programming Language." *The Bell System Technical Journal*, July-August 1978, pp. 1991-2019.

[10] Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1978.

[11] Harbison, Samuel P., and Guy L. Steele Jr. *C: A Reference Manual*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984.

[12] Purdum, Jack. *C Programming Guide*. Indianapolis: Que Corporation, 1983.

[13] Hancock, Les, and Morris Krieger. *The C Primer*. New York: McGraw-Hill Co., 1982.

[14] Plum, Thomas. *Learning to Program in C*. Cardiff, N.J.: Plum Hall Inc., 1983.

MS-DOS

[15] Norton, Peter. *The Peter Norton Programmer's Guide to the IBM PC*. Bellview, Wash.: Microsoft Press, 1985.

[16] Duncan, Ray. *Advanced MS-DOS*. Redmond, Wash.: Microsoft Press, 1986.

[17] Angermeyer, John, and Kevin Jaeger. *MS-DOS Developer's Guide*. Indianapolis: Howard W. Sams & Co., 1986.

[18] DeVoney, Chris. *MS-DOS User's Guide*. Indianapolis: Que Corporation, 1984.

Compiler Writing

[19] Calingaert, Peter. *Assemblers, Compilers, and Program Translation.* Potomac, Md.: Computer Science Press, Inc., 1979.

[20] Hansen, Per Brinch. *Brinch Hansen on Pascal Compilers.* New Jersey: Prentice-Hall, 1985.

[21] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Reading, Mass.: Addison-Wesley Publishing Co., 1986.

[22] Hunter, Robin. *Compilers: Their Design and Construction Using Pascal.* New York: John Wiley & Sons, 1975.

[23] Schreiner, Axel T., and H. George Friedman, Jr. *Introduction to Compiler Construction with UNIX.* Englewood Cliffs, N.J.: Prentice-Hall, 1985.

[24] Tremblay, Jean-Paul, and Paul G. Sorenson. *The Theory and Practice of Compiler Writing.* New York: McGraw-Hill Book Co., 1985.

[25] Tremblay, Jean-Paul, and Paul G. Sorenson. *An Implementation Guide to Compiler Writing.* New York: McGraw-Hill Book Co., 1982.

Index

! (logical-NOT operator) 90, 97, 357, 358, 361
!= (not-equal operator) 19, 90, 102, 345
19, 279
#...# 279, 280
#asm 130, 283, 298, 301, 309
#define 19, 125, 127, 136, 169, 201, 250, 261, 283, 298, 301, 551
#define FILE 180
#else 127, 128, 290, 291, 553
#endasm 130, 283, 301
#endif 127, 128, 290, 291, 553
#if... 290, 291, 553
#ifdef 127, 128, 289, 290, 291, 553
#ifndef 128, 289, 290, 291, 553
#include 19, 25, 26, 128, 129, 135, 179, 183, 301, 405, 554
% (modulo operator) 90, 99, 345
% (conversion-specification flag) 146, 148
%= (modulo-and-assign operator) 90, 109, 353
& (address operator) 49, 60, 82, 90, 98, 220, 338, 416
& (bitwise-AND operator) 47, 90, 102
&& (logical-AND operator) 90, 103, 341, 342
&= (bitwise-AND-and-assign operator) 90, 109, 353
' (apostrophe) 19
'\n' (newline character) 139
() (parentheses) 70, 73, 90, 312, 327, 330, 333, 356, 363
* (indirection operator) 49, 54, 60, 73, 79, 90, 98, 220, 222
* (multiplication operator) 90, 99
*= (multiply-and-assign operator) 90, 109, 353
+ (addition operator) 90, 100, 345,
++ (increment operator) 28, 90, 97, 357, 359, 360
+= (add-and-assign operator) 90, 109, 351, 353
, (comma) 300, 339
- (command-line switch) 196, 294

- (subtraction operator) 90, 100
- (unary-minus operator) 90, 98
-- (decrement operator) 90, 98, 357, 358, 359, 361, 353
-= (subtract-and-assign operator) 90, 109, 353
-A switch (alarm) 196, 294
-L# switch (listing) 196
-M switch (monitor) 196, 294
-NA switch (no alignment) 413
-NO switch (no optimizing) 165, 196, 294
-NSP switch (no stack probe) 415, 417
-P switch (pause) 196, 294, 549
/ (division operator) 90, 99, 332, 345
/= (divide-and-assign operator) 90, 109, 353
: (colon) 309, 310, 323, 325, 356
; (semicolon) 44, 300, 301, 304, 308, 310, 312, 320, 324, 332
< (less-than operator) 23, 24, 25, 28, 90, 101, 345
< (redirection of stdin) 188, 189
<< (left-shift operator) 90, 100, 345
<<= (left-shift-and-assign operator) 32, 90, 110, 353
<= (less-than-or-equal operator) 32, 90, 101, 345
<filename> 19
<l> 279, 280
<m> 279, 280
<n> 279, 280
<stdio.h> 129
= (assignment operator) 19, 25, 28, 66, 90, 109, 331, 351, 353, 354, 355
= (initializer) 299
== (equal operator) 19, 90, 102, 345
> (greater-than operator) 90, 101, 345
> (redirection of stdout) 188, 189
>= (greater-than-or-equal operator) 90, 101, 345
>> (redirection of stdout, concatenated) 188, 189
>> (right-shift operator) 90, 101, 345
>>= (right-shift-and-assign operator) 90, 110, 353
?...?...? 276, 279, 280, 418
?: (conditional operator) 90, 105, 355

[] (square brackets) 28, 57, 90, 295, 363
\ (escape character) 31, 37, 38, 375
\b (backspace character) 37, 375
\f (form-feed character) 37, 375
\n (newline character) 19, 66, 37, 278, 375
\ooo (octal character) 37
\t (tab character) 19, 37, 375
^ (bitwise-exclusive-OR operator) 90, 102, 345
^= (bitwise-XOR-and-assign operator) 90, 110, 353
_ (underscore) 36, 40
__eq (equal to) 212, 234
__ge (greater than equal to) 212
__gt (greater than) 212
__le (less than equal to) 212
__lneg (logical negation) 212, 230
__lt (less than) 212
__ne (not equal to) 212, 239
__switch 212, 213, 240, 321
__switch() 251, 252
__uge (unsigned greater than equal to) 212
__ugt (unsigned greater than) 212
__ule (unsigned less than equal to) 212
__ult (unsigned less than) 212
_abort() 416
_alloc() 414
_bdos2() 164
_ccargc 212
_getkey() 162, 163
_hitkey() 162
_main() 210, 211, 261, 294
_memptr 211
_n (numeric label) 315
_pop 381, 386, 390
_probe() 416, 417
{ } (braces) 19, 23, 25, 66, 67, 112, 309
| (bitwise-inclusive-OR operator) 90, 102, 345, 384
|= (bitwise-OR-and-assign operator) 90, 109, 353
|| (logical-OR operator) 19, 90, 104, 341, 342
~ (one's-complement operator) 90, 97, 357, 358, 361
80186 CPU 427
80188 CPU 427
80286 CPU 427
80386 CPU 427
8086 CPU 427
8086 family processors 427
8087 numeric data processor 422

8088 CPU 427

A

abort() 120, 142, 160, 190, 416
abs() 162
absolute address 427
absolute value 162, 374
actual address 223
actual argument 76, 304, 407
ADD12 269, 273, 333, 345, 364, 381
ADD1n 272, 274, 387
ADD21 272, 274
ADD2n 272, 274
ADDbpn 272, 274
addition 46
addlabel() 256, 324, 325, 411
ADDm_ 272, 274
addmac() 250
address 29, 49, 93, 94, 96
address, arithmetic 53, 61, 222
address, attribute 54
address, calculation of 430
address, difference 53, 54, 62, 70, 93, 94, 352
address, displacement 53, 62, 93, 94, 97
address, effective 51, 170
address, expression 80, 357
address, fix-up 193
address, function 229, 362, 363
address, null 55, 357
address, operator 49, 362, 550
address, physical 193, 222, 431
address, return 226
addresses, sum of 353
ADDSP 263, 269, 274, 313, 314, 368, 393, 411, 413
addsym() 257, 301, 313, 370, 409
addwhile() 257, 258, 317, 403, 404
ADDwpn 272, 274
AF (auxiliary flag) 432
AH 167, 211, 215, 432
aid 304
AL 167, 170, 211, 215, 220, 432
alarm 294
algorithm 26, 111
alignment, byte 166
alignment, double word 166, 418
alignment, function arguments 166
alignment, global variables 166
alignment, local variables 166
alignment, word 166
alpha() 286
alphabetic 27, 158, 286

alphanumeric 287, 288, 289
alphanumeric comparison 299
amatch() 284, 288, 299
ampersand 338
an() 286
AND, bitwise 96
AND12 269, 274, 345
ANEG1 269, 274, 358
angle brackets 179
apostrophe 20, 27, 29, 35, 38, 374, 553
AR.C 404, 414
argc 161, 181, 182, 187
ARGCNTn 269, 274, 368
argcs 294
argnext 409
argstk 303, 304
ARGTSZ 408
argtype 409, 410
argument 26, 74
argument, actual 72, 75, 76, 78, 79, 84, 86, 304, 407
argument, address/pointer 75, 75
argument, array 75, 75
argument, count 367, 559
argument, count routine 211
argument, declaration 23, 76, 224
argument, dummy 72, 75
argument, evaluation, order of 180
argument, expression 75, 76, 79, 80, 96, 226
argument, formal 72, 74, 75, 76, 78, 80, 406, 549, 554, 555
argument, function name 176
argument, list 76, 23, 31, 96
argument, name 23
argument, number of 76, 80, 81, 209, 407, 555
argument, offset to 84
argument, order of passing 176
argument, passing 23, 76, 78, 80, 84, 226
argument, references 226
argument, size 75
argument, string 75, 75
argument, type of 23, 76, 80, 407
argument, variable number of 176
argv 161, 181, 182, 187
argvs 294
Arithmetic and Logical Library 209
ARRAY 301, 312
array 21, 26, 39, 45, 57, 61, 97, 171, 254, 549, 552
array, address of 59, 369, 370
array, argument declaration 58
array, bounds 58, 61
array, character 21, 36
array, declaration 57

array, declared as global pointer 171
array, declared locally 171
array, dimension 31, 57, 58, 301, 312, 335
array, element 49, 57, 89, 92
array, external declaration 58
array, global 232
array, initial value 59
array, integer 21, 49, 59, 60, 97, 220, 222, 365, 369, 370
array, name 550
array, of pointers 180, 300, 301
array, pointer 21
array, reference 223
array, single dimension 177
array, size of 57, 58, 59, 66, 553
array, subscript 31
array, undimensioned 58
array, uninitialized and undimensioned 301
array, unsubscripted name 89, 93
arrays and pointers 59
ASCII 27
ASCII character set 159, 160
ASCII code 373
ASCII file 266
ASCII I/O functions 139, 140
ASCII string 265, 267, 274, 251
ask() 294, 295
ASL12 269, 275, 345
ASM file 196, 295, 402, 413
ASR12 269, 275, 345, 399, 400
assemble 187
assembler 22, 25, 195, 424
assembler, compatibility 191
assembler, instruction 273, 250
assembler, tasks 194
assembly language 114, 125, 130, 157, 215, 301, 377, 416
assembly-code file 191
assignment 24, 179
assignment, multiple 107
assignment, operator 70, 107, 179, 237, 331, 353, 354
assignment, order of evaluation 178
assignment, statement 41, 65, 70
associativity 89, 91
ASSUME directive 217, 218, 260
asterisk 181
astreq() 287
atoi() 154
atoib() 154
atol() 421
atolb() 421
auto 184
AUTOEXT 255, 260, 302

AUTOMATIC 255, 303
automatic storage class 39, 40, 66
AUX 189
auxbuf() 153
auxiliary buffer 153
auxiliary flag 432
auxiliary keystrokes 163
avail() 53, 84, 160, 190, 416, 417
AX (primary register) 39, 40, 41, 54, 71, 164, 167, 168, 172, 211, 212, 213, 214, 215, 216, 217, 220, 221, 222, 223, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 234, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 321, 325, 332, 333, 343, 400, 401, 432

B

back end 377
back end functions 259
backslash character 375, 396, 397
backspace 139, 375
base pointer 310, 428
batch file 140, 165, 190
BDOS 164
BELL 136
binary conversion specification 180
binary I/O functions 139, 140
binary operation 46, 55, 62, 351, 354, 355, 382
binary operator 234, 334, 336, 343, 346, 351, 419, 420
binary operator, signed 344
binary operator, unsigned 344
binary transfer 145
BIOS 163, 164
bitwise operations 96, 96
bitwise-AND operator 380
bitwise-OR operator 234
blanks() 286, 288
block 23, 24, 25, 26, 29, 40, 42, 113, 308, 313, 314, 315, 552
block, entry to 113
block, head of 42, 113
block, leaving 43
block, locals 411
block, nested 25, 26, 42, 324
block, subordinate 25
body of function 23, 24, 26
Boolean value 95
boundary, byte 166
boundary, double-word 432
boundary, paragraph 428
boundary, word 166, 167, 432
BP (stack-frame pointer) 40, 41, 84, 168, 224, 226, 228, 255, 304, 310, 338, 369, 403, 416, 417, 428, 431

brackets 222
break statement 117, 120, 121, 122, 181, 309, 322, 240, 322, 403, 554
bseek() 151
btell() 151
buffers, allocating storage for 293
bump() 286, 288, 292, 325, 341
BX (secondary register) 54, 164, 170, 172, 209, 212, 214, 215, 222, 232, 332, 333, 418, 432
BYTE 262
byte boundary 166, 412
BYTE_ 269, 275
BYTEn 269, 275, 412
BYTEr0 269, 275

C

C, filename extension 295
C, program 19
calc() 349, 352
call instruction 22, 24, 84, 431
CALL module 83, 218, 230, 234, 239, 259
CALL routines 209
call, by reference 78, 80
call, by value 78, 80
call, function 71
CALL.ASM 423
CALL1 269, 275, 368
callfunc() 364, 366, 367, 410, 412, 413, 418
CALLm 269, 275, 368
calloc() 52, 160, 169, 190, 293
cancel program execution 189
carriage return 27, 37, 138, 139, 266
carry flag 164, 432
case 322
case expression 322
case prefix 30, 115, 181, 117, 212, 214, 240, 309, 321, 322, 554
case sensitivity 159, 294
case, value of 251
cast operator 420, 422
causes 143
CBW 167, 170, 220, 222
CC.EXE 194, 199, 201
CC.H 199, 201, 371, 396, 404, 418, 420, 550, 551, 554
CC1.C 293, 396, 404
CC4.C 201, 259, 261, 274, 296, 379, 380, 381, 383, 384, 388, 389, 414, 415
ccargc() 83, 177, 211, 368
CCC.BAT 201
cdecl 181
CF (carry flag) 432
ch 284, 286

char 19, 44, 45, 77, 181, 298, 299, 304, 308, 312, 362, 402
char * 362
character 39, 174
character, address 54, 94, 220
character, alphabetic 27
character, argument reference 226
character, array 21
character, classification functions 158, 558
character, constant 20, 31, 27, 47, 66, 92, 289, 371, 374, 553
character, control 27
character, definitions 218
character, escape 31, 37
character, pointer 21, 94
character, promotion to integer 93, 169, 177, 178, 222
character, range 47
character, special 27, 31
character, string 49, 289
character, translation functions 159
character, variable 21, 46, 47
character-stream I/O functions 139
characters 173
CHR 253, 300, 308, 402
chrcon() 371, 372, 374
CL (argument-count register) 177, 211, 229, 368
CLASS 253, 303, 411
class field 254, 255
clearerr() 153
clearstage() 250, 251, 262, 264, 265, 335, 339, 364, 365, 389
CLIB.LIB 195, 200
closing files 142
code generated by a complex expression 237
code generated by addition operator 234
code generated by address operator 233
code generated by an IF statement 238
code generated by an IF/ELSE statement 239
code generated by assignment operators 237
code generated by constant expressions 217
code generated by direct function calls 229
code generated by division and modulo operators 233
code generated by DO/WHILE statement 245
code generated by equality operator 234
code generated by external declarations 223
code generated by external references 223

code generated by FOR statement 243
code generated by FOR without expressions 244
code generated by function arguments/references 228
code generated by global objects 219
code generated by global references 221
code generated by GOTO statement 245
code generated by increment prefix 231
code generated by increment suffix 231
code generated by indirect function calls 230
code generated by indirection operator 232
code generated by local objects/references 225
code generated by logical-AND operator 236
code generated by logical-NOT operator 231
code generated by non-zero and zero tests 239
code generated by SWITCH statement 241
code generated by WHILE statement 242
code generation 27, 259
code segment 52, 175, 223, 259, 260, 261, 428
code[] 273, 295, 378
CODESEG 260
coercion operator 422
colon 29, 30, 115, 117, 261, 310, 323, 325, 356
COM1: 136, 137, 138, 189, 269, 275, 358
comm 382, 390
comma 29, 30, 300, 339, 553
COMMAn 272, 275
command-line argument 138, 161, 181, 182, 183, 210, 246
command-line switches 294
comment 19, 20, 32, 289, 290
comment, length 20
comment, nesting 20
comment, placement 20
commutative operation 351, 379
COMMUTES 380
compare, unsigned 34
compile 185
compiler, bugs 201
compiler, command-line arguments 194
compiler, file redirection 194
compiler, filename 194
compiler, generations of 396
compiler, input and output 194
compiler, keyboard input 195, 215

compiler, parts of 25, 200
compiler, reliability of 202
compiler, revisions to 202
compiler, run-time options 194
compiler, source listing output 195
compiler, testing 201
compiler-relative stack pointer 257, 264, 351, 412
compiling Small C programs 191
compiling the compiler 199
compound statement 23, 25, 76, 111, 113, 113, 255, 256, 308, 309, 313, 322, 553
compound statement, nesting 23, 76, 313
compound() 255, 298, 308, 309, 313, 314
conditional compiling 21, 125, 127, 290
conditional operator 95, 355, 356, 551
console 153
const 181
constant 27, 33, 89, 249, 283, 301, 327, 550
constant, character 20, 33, 35, 47, 371
constant, decimal 34, 35
constant, expression 66, 68, 169, 170, 171, 300, 549, 552
constant, hexadecimal 35
constant, integer 362
constant, loading of 354
constant, numeric 27, 33, 371
constant, octal 34
constant, string 28, 31, 33, 371, 372
constant, subexpressions 264
constant, two-character 20
constant, unsigned 33
constant, zero 350, 357
constant, zero subscript 171
constant() 370, 371, 372, 373
constexpr() 300, 328, 330, 335, 336
continue statement 121, 122, 181, 309, 403, 554
control characters 159
control string 145, 147
control, flow of 24
control, transfer of 30
control-@ 163
control-BREAK 190
control-C 139, 139, 162, 192, 196, 267
control-S 162, 190, 195, 196, 267
control-X 140
control-Z 140, 143, 188
conversion specification 46, 47, 146, 148, 150
cooked data 139, 140, 188
cptr 250, 257, 301, 302
CPU flags 234

CR 136
CRIGHT1 293
CS (code-segment register) 52, 427, 428
cseek() 151
cseekc() 152
csp 257, 264, 313, 314, 325, 348, 351, 411, 412, 413
CSYSLIB 162, 210, 211, 261
CSYSLIB.C 261, 414
ctell() 152
ctellc() 152
cursor 140
CWD 333
CX (data register) 164, 177, 214, 418, 432

D

dangling reference 41
data bus 166, 168, 412, 418, 427, 432
data conversion 419
data segment 52, 163, 216, 223, 259, 261, 428
data segment register 210
data structure pointers 247
data structures 247
data types, limitations on 173
data/heap/stack segment 52, 52, 210
DATASEG 259, 260
DB (define-byte directive) 39, 219, 249, 255, 261, 266, 302, 305
DBL1 269, 364
DBL2 269, 273
DD (define double word) 418
DEBUG 211, 414
DECbp 272, 275
decimal 33
decimal number 373
decimal string 155, 156, 277
decl() 304, 312
declaration 19, 22, 300
declaration, argument 23
declaration, function 31
declaration, global 25, 19, 21, 26
declaration, in nested block 113
declaration, in switch statement 113
declaration, list 553
declaration, local 26, 310, 553
declaration, nesting 22
declaration, unsupported 21
declarator 23, 26
declare 45
declared 313, 314
declarg() 409
declloc() 255, 256, 297, 308, 310, 311, 312, 313, 409, 410, 412, 411, 413
decrement operation 182, 361

decrement operator 101, 360
DECwp 271, 273
default drive 190
default filename extensions 198
default label 322, 323
default prefix 30, 115, 116, 184, 214, 216, 239, 309, 321, 322, 552, 554
define 45
Defined More Than Once 195
definition 22
definition, macro 126
DEL 140, 159
delete character 159
delete key 140
delete, file 152
delwhile() 258, 317, 404
destination index 432
device 136, 153, 188
DF (direction flag) 432
DI 432
differences between Small C and full C 173
digit 27, 28
digit, hexadecimal 35
dim 300, 301
dimension 57
direct reference 354
direction flag 432
directory 179
disk file 153, 188
disk head movement 153
disk space 168
DISOPT 201, 261, 265, 296, 380, 389, 414
DIV12 263, 269, 275, 332, 345
DIV12u 263, 268, 269, 275, 345
divide 94, 101, 263
do statement 121, 181, 243, 257, 308, 309, 323, 550, 554
doargs() 303, 304, 409
doasm() 301, 308, 309
dobreak() 258, 309, 323, 403, 404
docase() 309, 322
docont() 258, 309, 323, 403, 404
dodeclare() 299, 402, 403, 418, 419
dodefault() 309, 323
dodefine() 301
dodo() 258, 308, 309, 318, 403
doexpr() 308, 310, 325, 328, 330, 333, 335, 392
dofor() 258, 308, 309, 319, 403
dofunction() 248, 256, 302, 304, 307, 308, 309, 402, 406, 409, 410, 416, 418, 419
dogoto() 309, 324, 411
doif() 308, 309, 315, 330
doinclude() 301, 405

dolabel() 309, 310, 325, 411
doreturn() 309, 325, 419
doswitch() 252, 258, 308, 309, 320, 322, 323, 403
dotype() 403, 417
double() 348, 350, 351, 421
double-word aligned 170
dowhile() 258, 308, 309, 317, 318, 403
down() 328, 329, 330, 331, 332, 334, 335, 341, 343, 344, 345, 346, 357
down1() 328, 329, 330, 331, 332, 334, 341, 342, 343, 345, 348, 349, 352, 354, 355, 357
down2() 328, 329, 330, 332, 333, 334, 335, 343, 344, 345, 346, 349, 350, 351, 352, 354, 355, 365, 399, 400, 419, 420
downward compatibility 173
drive designator 189
dropout() 342, 343, 345, 355, 356, 402
DS (define-storage directive) 52, 210, 427, 428, 429, 430, 431, 432
dtoi() 154
dtol() 421
dumplits() 249, 265
dumpstage() 264, 265, 382, 383, 388, 389, 390, 392, 414, 415
dumpzero() 266, 302
DUP operator 219
DW (define-word directive) 39, 219, 249, 255, 261, 265, 302
DX (data register) 166, 233, 332, 333, 432, 418
dynamic memory allocation 255

E

echo, of rubout 140
echoed data 139
effective address 168
efficiency considerations 165
efficiency, effect of parentheses 169
efficiency, integer and character fetch 166, 167
efficiency, integer and character store 171
element 57
else 116, 181, 315
END directive 210, 261, 296
end of file 139, 143, 143, 152, 188
ENDS directive 217, 218, 260
endst() 300, 303, 312
ENTER 269, 275, 417
ENTER key 188, 196, 549
entry point 77, 192, 193, 218, 301, 303
entry point, initial 209

enum 181
EOF 19, 136, 139, 143, 143, 144, 150, 151, 161, 182, 292, 301, 405, 557
eof 296
EQ10f 269, 275, 340, 342, 401
EQ12 269, 275, 340, 345
equality 46
ERR 136, 142, 151, 152, 153, 154, 157, 557
errflag 310
error code 190
error condition, file 152
error messages 549
error reporting 310
error() 410
ES (extra-segment register) 427, 428, 429, 432
escape sequence 37, 66, 375, 300, 396
escape sequence, octal 92
exclusive OR 278
EXE file 168, 185, 191, 218, 402
executable file 168, 185, 191
execution, beginning of 24, 26
execution, of Small C programs 185
exit code 139, 160, 161, 190, 416
exit() 120, 142, 190, 210
expression 22, 29, 89, 297, 316, 550
expression, analysis 327, 316
expression, analyzer 78, 92, 95, 251, 308, 327, 330, 357, 368, 377, 557, 558
expression, compile-time evaluation 95
expression, constant 66, 68
expression, constant or variable 95
expression, evaluation 31, 62, 89, 113, 263
expression, function 69
expression, list 30, 89, 115, 117, 122
expression, operators 31, 327
expression, primary 96, 97
expression, properties 105, 106, 336
expression, run-time evaluation 95
expression, statement 22, 70, 114, 220
expression, subscript 57
expression, value of 89, 95
expression, work done by 70, 112, 114
expression() 328, 330, 333, 335, 336, 339, 368
extern 22, 24, 25, 42, 44, 45, 181, 193, 298, 299
EXTERNAL 255, 298, 299

external 22, 24, 25, 39, 42, 44, 302
external declaration 223, 295, 299
external functions, undeclared 260
external objects 254
external reference 74, 178, 194, 195, 261
external() 262, 301
extra segment 428
EXTRN directive 25, 223, 262

F

far 182
far pointers 424
fclose() 142, 296
fd 142, 142, 143, 144, 145, 148, 150, 151, 152, 153
feof() 144, 152
ferror() 144, 144, 145, 153
fetch() 344, 356, 360, 364, 419
fetching 170
fgetc() 139, 143, 140, 557
fgets() 143, 282, 290
file 143, 192
file, assembly-code 195, 196
file, automatic close 164, 192
file, buffers 142
file, close 153
FILE, control structure 183
file, current position 142
file, descriptor 136, 138, 183
file, executable 193, 194
file, independence 137
file, library 195
file, object 193, 194, 195
file, open 139, 162, 183, 290, 291, 293
file, output 250
file, pointer 136, 150, 180
file, redirection 295
file, specification 140, 189
file, standard error 197
file-handle MS-DOS calls 153
file/device independence 138
filename 179, 197, 294, 301
filename, command-line argument 186
findglb() 256, 301
findloc() 256
flab1 317, 318, 332
flab2 318
flag register 432
float 181
floating point 422
fopen() 141, 142, 142, 301
for statement 120, 181, 242, 243, 257, 308, 309, 323, 550, 554
form feed 37, 375
formal argument 77, 303, 406, 549

format conversion functions 154
formatted output 145, 147
FORTRAN 182
forward reference, to function 174
fprintf() 83, 148, 180, 421
FPRINTF.C 177
fputc() 139, 144, 266, 267
fputs() 145
frame pointer 84
fread() 143
free memory 52, 160
free() 161
freopen() 142
front end 283
fscanf() 83, 150, 180, 421, 557
FSCANF.C 177
fseek() 151
full C, compatibility with 173
FUNCTION 300, 301, 303, 409
function 19, 22, 21, 24, 26, 28, 30, 45, 69, 253, 297, 327
function, address of 70, 94, 96, 174, 175, 327, 362, 367
function, algorithms 76
function, arguments 71, 166
function, automatic declaration of 175, 296
function, body of 26, 85, 111, 254, 304, 305, 309, 311
function, call 22, 31, 40, 41, 69, 70, 71, 73, 76, 77, 78, 80, 83, 84, 89, 91, 92, 94, 96, 178, 182, 304, 327, 363, 366
function, complexity of 69
function, concept of 70
function, declaration 21, 22, 23, 31, 42, 72, 73, 228, 308, 550, 553
function, declarator 26
function, definition 75, 71, 296
function, direct call 367
function, exit 41
function, indirect call 78, 78, 175, 367
function, label 367
function, name 23, 70, 94, 96, 178, 175, 327, 330, 363
function, pointer declaration 73
function, pointer to 73, 312
function, power of 69
function, predeclared 178
function, prototyping 407
function, recursive call 86
function, reference 254
function, return data type 73, 75
function, return from 69, 85
function, return value 71, 71, 77
function, side effects of 72

function, terminology 71
function, undeclared 25, 369
function, value of 69, 94, 96
functions and subroutines 70
fwrite() 144

G

gc 383, 384, 391
gch() 283, 284, 286, 288, 398, 411, 412, 420
GE10f 269, 274, 340
GE12 269, 274, 340, 345
GE12u 269, 274, 340, 345
gen() 250, 263, 264, 265, 266, 274, 316, 318, 324, 325, 332, 349, 350, 352, 356
generated code 217
get functions 139
getarg() 161, 186, 187, 292, 293
GETb1m 270, 274
GETb1mu 270, 274
GETb1p 263, 270, 274
GETb1pu 263, 270, 274
GETb1s 271, 274
GETb1su 271, 274
getc() 143
getchar() 19, 143
getint() 178
getlabel() 257, 315, 316, 330, 355, 356
getpop() 390, 391, 392
gets() 140, 144
GETw1m 270, 274, 386
GETw1m_ 271, 274
GETw1n 270, 274, 332, 342, 348, 354, 372, 399, 401
GETw1p 263, 270, 274
GETw1s 271, 274
GETw2m 271, 274, 386
GETw2n 270, 274, 348, 350, 364, 381
GETw2p 271, 274
GETw2s 270, 275
glbptr 257, 294
global 40, 42, 52
global data 261
global declarations 25, 27, 21, 26, 298, 299, 300, 402
global definitions 193, 220
global integer 168
global level 22
global name 29
global object 24, 25, 29, 72, 168, 210, 248, 261
global object, initializing of 294
global references 220
global symbol table 256
global variables, alignment of 166
go 383, 384
goto keyword 553

goto statement 29, 30, 113, 114, 120, 181, 245, 305, 309, 311, 314, 324, 411, 549, 554
grammar 297, 298
greater than 94
greater than or equal 94
grouping 91
GT10f 270, 276, 340
GT12 270, 276, 340, 345
GT12u 270, 276, 340, 345
gv 383, 384, 391

H

hashing algorithm 250, 256
header files 129, 130
header() 259, 295
heap 52, 160, 210, 211
hexadecimal 33, 155, 156, 373
high-level I/O functions 139
high-level parsing 297
HIGH_SEQ 388
hyphen 186, 294, 373

I

id 298, 312
IDENT 253, 303, 304
ident 301
identifier 28
identity field 253, 254
IDIV 233
IF (interrupt-enable flag) 432
if statement 115, 181, 308, 309, 315, 330
if...else... statement 171
ife 383, 385, 391
ifl 383, 385, 387, 391
iflevel 290, 291
ifline() 289, 290, 291
illname() 312
inbyte() 285
INCbp 272, 276
include file 292, 554
including source files 125, 129
increment 178
increment operator 97, 231, 360
INCwp 272, 276
index registers 432
indirect function call 175, 176
indirect reference 54, 354, 366, 369
indirection operator 54, 361, 366
inequality 46
inest 405
init() 301, 302
initial value 40, 41, 59, 67, 65, 66, 219
initializer 66, 68, 248, 549
initializer, character string 302
initializer, constant expression 302
initializer, string 552

initializing, local array 169
initializing, variables 293, 294
initials() 248, 301, 302, 412
inline() 248, 290, 291, 292, 301, 405, 406
input 292, 405
input file 153, 284, 285
input line 248
input/output functions 136
input/output operations 72
input/output redirection 188
input/output statements 123
input2 292, 301, 405
instruction pointer 84
instruction prefix 429
INT 254, 303, 308, 338, 352, 370, 371, 372, 374, 402
int 19, 22, 25, 28, 44, 45, 77, 184, 298, 299, 304, 312, 362
int * 362
integer 21, 22, 27, 34, 39, 166, 173, 174, 232
integer, address 220
integer, alignment of 166
integer, argument references 226
integer, array 59, 21
integer, data types 177, 178
integer, definition 218
integer, global 168, 169
integer, local 168, 169
integer, long 33
integer, pointer 21
integer, range 45
integer, signed 35, 45, 92, 94
integer, unsigned 92
integer, variable 21, 93
integers and pointers 53
inter-module reference 22
internal format 147, 148
interrupt-21 164
interrupt-enable flag 432
interruption of program execution 185, 189, 190
invoking the compiler 197
IP (instruction-pointer register) 84, 428, 431
is...() functions 558
is[] 336, 337, 339, 345, 346, 354, 357, 360, 361, 366, 369, 371
is[CV] 337, 338, 339, 349, 350, 352, 358, 359, 361, 362, 364, 366, 367, 370, 372, 373, 374, 419
is[OP] 337, 338, 340, 349
is[SA] 337, 338, 340, 348, 349, 350, 358, 361
is[ST] 337, 349, 353, 354, 359, 364, 365, 367, 369, 370
is[TA] 337, 338, 349, 353, 359, 362, 364, 366, 369, 370
is[TC] 337, 338, 339, 345, 349, 352, 354, 359, 364, 367, 370, 372, 373

is[TI] 337, 349, 353, 354, 358, 359, 361, 362, 364, 366, 369, 370
is1[] 356, 357
is2[] 336, 345, 346, 350, 356, 365, 558
is2[ST] 349
is2[TA] 349
is2[TC] 352, 363
is2[TI] 349
is3[] 354, 355, 356
isalnum() 158
isalpha() 158
isascii() 158
isatty() 153
iscntrl() 158
iscons() 153, 557
isdigit() 158, 159
isfree() 380, 390, 392
isgraph() 158
islower() 159
isprint() 159
ispunct() 159, 557
isspace() 159
isupper() 159
isxdigit() 158, 159
itoa() 154
itoab() 154
itod() 155, 156
itoo() 155
itou() 156
itox() 156

J

JMPm 270, 276, 316, 340, 411
jump, conditional 235
jump, instructions 431
jump, unconditional 235

K

keepch() 285, 290
keyboard 21, 136, 137, 138, 139, 140, 161, 162, 188, 189, 246, 389, 398, 402, 406, 413
keystroke, auxiliary 162, 163
keystroke, pending 162, 163
keystroke, waiting for 162, 163
keyword 19, 24, 28, 181, 283, 308
kill() 301

L

label 29, 30, 39, 41, 96, 114, 216, 234, 245, 253, 254, 255, 303, 310, 324, 325, 354, 355, 368, 370, 372, 403, 411, 549, 553
label, for literal pool 248
label, function 174
label, in symbol table 314, 315
label, name 279

label, number 216, 245, 252, 255, 315, 316, 323, 324, 355, 386
label/value table 322, 323
LABm 270, 276, 316, 318
language construct 27
lastst 308, 309, 313, 314
LE10f 270, 276, 340
LE12 270, 276, 340, 345
LE12u 270, 276, 340, 345
LEA 225
left() 156
length, of object 254
less than 95
less than or equal 95
level1() 327, 328, 330, 331, 333, 334, 336, 339, 345, 351, 353, 355, 363, 365, 368
level2() 327, 328, 330, 331, 355, 557
level3() 328, 331, 341, 355, 357
level4() 328, 331
level5() 328, 331, 345
level6() 328, 331, 345
level7() 328, 345
level8() 328, 345
level9() 329, 345
level10() 329, 345
level11() 329, 331, 333, 345
level12() 329, 331, 332, 341, 345, 357
level13(327, 329, 331, 332, 351, 357, 358, 359, 360, 361, 366, 367, 422
level14() 327, 329, 331, 346, 351, 354, 359, 362, 363, 364, 366, 367, 370
lexcmp() 159
lexicographical comparison functions 159
lexorder() 160
LF 136
library 25, 29, 123, 191, 193, 209
library, functions 135
library, manager 193
library, Small C 176
library, standard Small C 135
line 19, 248, 284, 286, 289
line buffers 293
line feed 27, 37, 138, 139, 266
line size, maximum 396
line, maximum length 551
line, preprocessor 21
LINEMAX 396, 573
LINESIZE 247, 396, 551
link 25, 187
linker 22, 25, 42, 44, 135, 178, 193, 195, 201, 202, 220, 225, 376, 415
linker, errors 195
linker, tasks 194
list 30

listfp 290
listing, of source 290
LITABSZ 247, 551
litchar() 300, 374, 375
literal pool 218, 248, 265, 266, 277, 291, 299, 302, 303, 304, 307, 375, 376, 398
literal pool, label 304, 307, 376
literal queue 551
litptr 249, 302, 378, 398
litq 249
LNEG1 270, 274, 361
local 25, 42
local declaration 26, 76, 255, 309, 312, 313, 316, 327
local declarations and references 225, 226
local function declaration 310
local initializers 313
local integer 172
local object 24, 25, 29, 86, 172
local table 255
local variables 113, 229
local variables, alignment of 170
locptr 255, 256, 292, 316, 317
logical comparison routines 211, 213
logical operations 95
logical value 95
logical-AND operator 96, 235
logical-NOT operator 96, 231, 364
logical-OR operator 96, 235
LONG 419
long 184
long arithmetic 423
long int 419
long integers 33, 419
long variables 149, 151
loop control statements 117
low-level I/O functions 139
low-level parsing 304
lowercase 27, 28, 35, 161, 162, 198
LPT1: 136, 137, 138, 190
lptr 248, 282, 284, 325
LT10f 270, 274, 343
LT12 270, 274, 343, 346
LT12u 270, 274, 343, 346
ltoa() 423
ltoab() 423
ltod() 423
ltoo() 423
ltou() 423
ltox() 423
lvalue 61, 79, 224, 306, 344, 346, 356, 357, 360, 361, 362, 363

M

m1 386
m2 386, 389
m3 386
m4 386
macn 250
MACNBR 573
MACNSIZE 247
macq 250, 304
MACQSIZE 247, 574
macro 28, 250
macro name 126, 249, 287, 288, 304
macro name table 249, 288, 291, 304, 574
macro pool 249
macro processing 125, 130
macro replacement text 248
macro string queue 574
macro substitution 288
macro symbol 28
macro text 249, 250
macro text queue 249, 288, 291, 303, 304
magnitude bits 45
main() 19, 24, 26, 83, 142, 164, 172, 185, 189, 191, 192, 211, 247, 259, 260, 261, 291, 292, 293, 294, 399, 405, 408
malloc() 52, 163, 192
manifest constant 126
match() 281, 282, 286, 297, 314, 398, 407
mathematical functions 162
memory 22
memory, access, efficiency of 170
memory, address 429
memory, allocation 163, 172
memory, arrangement 429
memory, blocks 425
memory, dynamic allocation of 247
memory, free 52, 211
memory, insufficient 192, 211, 416
memory, management statements 123
memory, model 51, 211
memory, segmentation 429
memory, set-up of heap 211
MINIX 426
minus 350, 361, 374, 376
minus sign 27, 33
mline 248, 281, 287, 288
MOD12 263, 270, 274, 348
MOD12u 263, 270, 274, 348
module 22, 25, 211, 254, 299
module, assembly language 211
module, separately compiled 22
modulo 94
monitor option 294, 304
MOVE21 263, 270, 274, 332, 348, 383

msname[] 304
MUL12 271, 276, 348
MUL12u 271, 276, 348
multi-precision arithmetic 422
multidef() 306
multiply 94
multiply, fast 101

N

NAME 253, 305
name 19, 25, 28, 141, 571, 573
name, field 256
name, global 29
name, length 28
name, macro 289
name, superseded 25
name, undeclared 42, 73
names 287
nargs 414
nch 286
ncmp 316
NDP 424
NE10f 271, 276, 342, 343, 344, 402
NE12 271, 276, 342, 348
NEAR 262
near 182
near pointers 426
NEARm 271, 276
need() 339, 407, 410
needsub() 301, 315
neg 385, 387, 389, 393
nested block 327
nesting limit 572
NEWLINE 136, 139
newline 37, 139, 142, 143, 144, 265, 266, 378
newline() 266, 267
newseg 260
next 392
nextop() 288, 344, 346
NO 136
NOALIGN 415
NOCCARGC 581
nogo 413
non-zero 95
nosign() 355
not 576
NOTICE.H 200, 293
ns() 300
NULL 136, 142, 142, 143, 150, 151, 152, 158, 161
null address 55, 360
null character 142, 143, 144, 154, 156
null statement 312
null string terminator 158
number() 375, 376, 373, 421
numeric 286
numeric characters 159
numeric constant 92, 374, 375
NUMGLBS 253, 572
NUMLOCS 253, 573

O

OBJ files 200, 416
object 22, 39, 42, 255
object code 297
object file 191
object module 41, 42, 135, 193
octal 33, 154, 155
octal character 37
octal escape sequence 92, 178, 278, 381
octal number 376, 378
OF (overflow flag) 435
offset 175, 253
OFFSET 305, 306, 408
OFFSET operator 216, 220
offset symbol 411
oldseg 259, 260
one's complement 363
op[] 288, 345, 348
op2[] 288, 345
open modes 141
openfile() 292, 295
opening files 142, 295
oper 348, 351, 352, 353, 354, 357
oper2 349, 351, 354
operand 89, 90
operand, addressing modes 432
operand, attributes 94, 348, 349
operand, computed 94
operand, count 89
operand, primary 92, 93
operand, reference 429
operand, unsigned 34
operating system 24
operation 143
operation, binary 46, 47
operation, signed 46, 47, 94, 351
operation, unsigned 46, 47, 62, 94, 353
operator 27, 89, 90, 283
operator, binary 34, 94
operator, multi-character 19
operator, precedence 360
operator, properties 89
operator, unary 34, 97
operator, value of 95
opindex 341, 344
opoff 344
opstr 344
optimize 294, 364
optimized output code 165
optimizer 228, 234, 236, 268, 579
optimizing 250, 265, 379
optimizing, adding cases 389
optimizing, attempts 391
optimizing, frequency counter 383, 392
optimizing, instructions 382, 384
optimizing, language 389

optimizing, metacodes 383, 385, 389
optimizing, optimization phase 384, 385, 392
optimizing, potential cases of 295
optimizing, recognition pattern 382
optimizing, recognition phase 384, 392
optimizing, statistics 201, 296
options, setting of 291
OR, exclusive 96
OR, inclusive 96
OR12 270, 276, 345
otoi() 154
otol() 421
outcode() 250, 251, 264, 265, 268, 274, 276, 277, 278, 279, 280, 391, 413
outdec() 267
outline() 267
outname() 267, 277
output file 138, 144, 145, 153, 250, 261, 263, 264, 265, 267, 278, 292, 295, 296, 315, 325, 339, 380, 391
output functions 259
outsize() 262
outstr() 267
overflow flag 432

P

p-code 250, 251, 263, 264, 265, 267, 268, 379, 383, 386
p-code, associated value 263, 264, 268, 276, 277, 380, 386
p-code, commutative 384
p-code, double-word precision 421
p-code, names 558
p-code, new 420
p-code, property byte 381, 382, 394
p-code, translation 274, 275, 380, 384
p1 384
p2 384
p3 386, 389
p4 386
pad() 158
paragraph 428
parallel port 136, 138
parentheses 23, 29, 31, 60, 90, 91, 94, 96, 97, 176, 332, 336, 341, 366, 367, 370, 371
parenthesis 407, 410, 424, 553
parity flag 432
parse tree 297
parse() 295, 297, 298, 299, 300, 303, 304, 420
parser 283, 284, 297
parsing 248, 289

parsing, function definitions 305
parsing, methods 297
parsing, preprocessor directives 303
pascal 182
path 189
PATH environment variable 200
pause 294
pause program execution 162, 189
peep() 250, 251, 265, 295, 384, 385, 386, 387, 388, 389, 390, 391, 392, 416, 421
peephole optimizer 250, 377
performance criteria 165
period 27
PF (parity flag) 432
pfree 383, 388, 392
physical address 431
pline 248, 284, 285, 289, 290
plus sign 27, 33
PLUSn 272, 276
point() 262, 300
POINT1l 270, 276, 372
POINT1m 270, 276, 359, 370, 383
POINT1s 271, 276, 369, 388, 389
POINT2m 272, 277, 272
POINT2m_ 277
POINT2s 272, 277, 388, 389
POINTER 304, 312
pointer 21, 26, 39, 45, 49, 60, 61, 89, 158, 166, 171, 173, 231, 247, 253, 293, 301, 302, 303, 332, 340, 349, 353, 356, 368, 369, 371, 372, 373, 374, 390, 410
pointer, address of 54
pointer, argument 61
pointer, arithmetic 49, 53
pointer, array 29, 53, 175
pointer, chain 49
pointer, character 29, 34
pointer, character string initialization 262
pointer, comparison 55
pointer, declaration 49, 50, 302
pointer, efficiency of reference 172
pointer, far 426
pointer, global 232
pointer, integer 21
pointer, name 97, 222, 365
pointer, near 426
pointer, reference 222
pointer, size of 50, 51
pointer, subscripted 61, 550, 222
pointer, type of 50
pointer, uninitialized 302
pointer, value of 54
pointers and integers 53

POINTm 364
poll() 140, 161, 163, 190, 267, 557
pop instructions 431, 434
POP2 264, 271, 277, 351, 383, 385, 388, 389, 392, 394
portability 173
precedence, parentheses 96
preprocess 290
preprocess() 250, 284, 285, 286, 287, 288, 295
preprocessing 299
preprocessor 20, 21, 125, 248, 283, 284, 290, 291, 426
preprocessor directive 19, 20, 21, 26, 29, 125
preprocessor line 21
PRI 382, 394
primary expression 97, 97, 176
primary input file 292, 301, 290
primary operand 92, 93, 371
primary register 77, 97, 114, 209, 211, 215, 263, 269, 372, 373, 375, 376, 381-4, 389, 392, 394, 420
primary() 329, 333, 334, 336, 337, 338, 357, 366, 367, 368, 371, 374, 375, 376, 422
primitive functions 162
printer 196, 266
printf() 82, 83, 146, 147, 147, 176, 177, 423, 425, 533, 534, 581
printf(), conversion specifications 180
PRN 189
procedure 22, 70
program control functions 160
program control statements 123
program control, overall 293, 295
program loading 65
program modules 191, 193, 194
program name 182
program size 165
program speed 165
program structure 19
program translation 191, 210
programming style 165
project, arrays of pointers 426
project, better code for integer address differences 399
project, better code for logical AND and OR operators 401
project, cast operator 422
project, consolidated type recognition 403
project, continued character strings 396
project, eliminate quote[] 396

project, eliminating while queue 404
project, floating point numbers 424
project, fseek() and ftell() 425
project, full access to DOS facilities 425
project, local data initializers 426
project, long integers 419
project, math functions 425
project, memory management routines 425
project, multiple memory models 425
project, nested comments 426
project, nesting include files 405
project, port to another processor 426
project, port to other operating systems 426
project, prototype function declarations 407, 408
project, restarting the optimizer loop 416
project, separate stack segment 425
project, sprintf() and sscanf() 425
project, stack probes 417
project, test programs 396
project, void argument lists 407
project, word alignment 413
prototype function declaration 73
pseudo-code 250, 267
PUBLIC 261
PUBLIC directive 25, 77, 218, 219
public() 261, 262, 301
punctuation 27, 29, 283
push instructions 431
PUSH1 264, 271, 275, 388, 399, 420
PUSH2 272, 275
PUSHES 382-3
PUSHm 272, 277
PUSHp 272, 277
PUSHs 272, 277
put functions 139
PUT_m_ 272, 277
PUTbm1 271, 277
PUTbp1 271, 277
putc() 144
putchar() 19, 86, 116, 118, 144
putint() 174
puts() 145
PUTwm1 271, 277
PUTwp1 271, 277

Q

question mark 278

R

range, decimal 33
rather 153
raw data 139
rDEC1 271, 275, 389
rDEC2 272, 275
read functions 139, 140
read operation 141
read() 144
readwhile() 258, 404, 405
records 173
recursion 298, 308, 309, 317, 318, 322, 406
recursive algorithms 417
recursive call 86, 329
recursive descent 297
redirection of standard files 137, 185, 293
redirection specification 182, 186, 188, 189
reference, global 373
reference, indirect 373
REFm 271, 277
register 181
registers, Small C 209, 210
relational operators 101, 234, 236, 342
rename() 152
repeat prefix 158
reserved words 28, 29, 181
resolve reference 193
RET 312
RETURN 264, 271, 277, 328
return 24, 181
return address 84, 86, 226, 307
return instructions 431
return statement 85, 120, 122, 228, 312, 317, 328
return value 71, 72, 77, 78, 85, 97, 178, 228
returning control 22
reverse() 158
rewind operation 143
rINC1 271, 272, 277, 362, 363, 389
rINC2 273, 277
rubout 140
run-time library 424

S

savcsp 316
savloc 317
scaling 93, 94, 97, 172
scaling address differences 356
scaling address displacements 352, 353
scaling factor 54
scanf() 83, 148, 148, 149, 176, 177, 423, 425, 566, 568

scanf(), conversion specifications 180
scope of variables 43, 43, 256
screen 136, 137, 139, 188, 189, 194, 195, 196, 265, 267, 295, 397, 398
search() 250, 256, 257, 289, 290, 304
SEC 382, 394
secondary register 209, 215, 263, 269, 349, 353, 358, 368, 380, 382, 392, 394, 420, 426
seek operation 143, 150
SEGMENT directive 217, 218, 260
segment, address 51, 51, 429, 430
segment, code 51
segment, data 51
segment, data/stack 52, 52
segment, offset 51, 51, 429
segment, registers 51, 210, 211, 429
segment, size 51, 52
segment, stack 51
semicolon 29, 111, 112, 197, 292, 300, 301, 334, 369
seq 392
seq##[] arrays 382, 384, 385, 386, 387, 388, 389, 390, 392, 393
seq[] 295, 390, 391, 416
serial port 136, 138
serial reusability 65
setcodes() 274, 262, 263, 264, 295
setseq() 295, 390, 416
setstage() 250, 335, 363
SF (sign flag) 435
sfree 384, 392
short 181
SI 432
side effect 72, 79, 80, 333, 334, 338, 339, 350, 369, 370
sign 153, 154, 155
sign bit 35, 45, 46
sign extension 46, 168, 170, 179, 220, 335, 386, 393
sign flag 435
sign() 160
signed 44, 45, 47, 75, 92, 93, 181
signed comparison 101, 171
signed integer 55, 92, 94, 96
signed operation 46
signed p-code 349, 354
signon message, displaying the 293
simple statement 44, 76
SIZE 303, 408, 412
size 253, 301
size field 255
sizeof() 90, 98, 171, 184, 360, 361, 364, 525, 580
skim() 330, 332, 333, 344, 345, 346, 348, 360, 402

skip 280
skip() 363
skiplevel 290, 291, 292
slash 186, 332
slast 250, 380
sname[] 369
snext 250, 251, 262, 263, 264, 265, 350, 380, 383, 384, 385, 386, 389, 391, 392, 393
source code 21
source file 20, 24, 26, 42, 261, 292, 295, 298, 299, 558
source files, Small C 200
source index 435
source lines 290
SP (stack-pointer register) 40, 41, 84, 209, 212, 226, 228, 229, 264, 328, 414, 416, 418, 419, 431
SPACE 136
space 27
special character 27, 31, 160
special keystrokes 579
square brackets 31, 57, 97
SS (stack-segment register) 52, 210, 430
ssname 305
ssname[] 261, 298, 300, 301, 305
stack 29, 40, 52, 77, 80, 83, 84, 86, 114, 160, 187, 210, 215, 226, 226, 228, 229, 254, 306, 332, 334, 335, 336, 340, 350, 352, 353, 354, 358, 370, 371, 372, 381, 382, 383, 392, 394
stack frame 29, 40, 83, 168, 176, 226, 255, 372, 404, 414, 418
stack frame pointer 255, 340
stack frame, nested 84, 298
stack operations 428
stack overflow of 84, 160, 211, 417
stack pointer 84, 209, 210, 215, 226, 264, 431
stack probe 425
stack segment 52, 430
stack segment register 210
stack set-up of 210
stack space 332, 417, 419, 580
stage 380, 384, 391
STAGESIZE 247, 576
staging buffer 95, 250, 264, 265, 293, 333, 334, 338, 341, 348, 353, 354, 364, 366, 368, 380, 383, 384, 386, 387, 388, 389, 391, 392–94, 421
stail 265, 380, 391
standard auxiliary file 138
standard error file 137, 197
standard file descriptors 137
standard files 137, 138, 188
standard input file 137, 142, 148, 293

standard output file 118, 137, 144, 145, 261, 295, 296
standard printer file 138
start 350
start: label 210
start-up routine 209, 210, 425
STARTGLB 253
STARTLOC 253, 256
STASM 311
statement 19, 23, 111, 297, 309
statement, assignment 70, 70
statement, break 117, 120, 121, 122, 240
statement, call 22, 24
statement, compound 23, 25, 111, 113, 113
statement, continue 121, 122
statement, controlled 111, 238, 310
statement, controlling 111, 310
statement, declaration 111
statement, do 121, 244, 257
statement, executable 26, 42, 310, 312
statement, expression 22, 114
statement, for 120, 242, 243, 257
statement, goto 29, 30, 114, 120, 245
statement, if 19, 25, 115
statement, if...else... 171
statement, nesting 23, 257, 310
statement, null 112
statement, procedural 111
statement, return 85, 120, 122, 228
statement, sequence 23, 310
statement, simple 23, 29, 44, 111, 115, 116, 171
statement, switch 30, 240, 241
statement, terminator 29
statement, while 19, 28, 118, 257
statement() 252, 298, 303, 307, 310, 312, 314, 315, 333, 337, 403, 413, 414, 415, 418, 420
STATIC 255, 299, 305
static 25, 39, 40
static declaration 299
static storage class 168, 181
STBREAK 311
STCASE 311
STCONT 311
stdaux 137, 137, 138, 180
STDEF 311
stderr 136, 137, 180, 189, 265
stdin 136, 137, 142, 144, 150, 180, 180, 188, 189, 194, 295
stdio.h file 135, 139, 180, 180, 200

STDO 311
stdout 136, 137, 147, 180, 188, 189, 194, 201
stdprn 137, 137, 138, 180
step() 361, 363
STEXPR 311
STFOR 311
STGOTO 311, 316
STIF 311
STLABEL 311
storage class 255, 260
storage class, automatic 39, 40, 66
storage class, external 39, 42, 299
storage class, static 39, 40, 168, 299
store() 356, 358, 363, 421
stowlit() 249, 378
strcat() 156, 157
strchr() 158
strcmp() 157, 159
strcpy() 157
streq() 87, 283, 287–89, 379
STRETURN 311, 312
string 20, 28, 31, 36, 67, 156, 157, 158, 216, 248, 261, 262, 263, 302, 337, 344, 360, 371, 374, 375, 376, 377, 378, 379, 380, 381, 392, 396
string, address 302, 374
string, comparison 157, 287
string, constant 96, 374, 375, 376
string, initializer 561
string, instructions 435
string, length of 158, 568,*
string, long 37
string, manipulation functions 156
string, reference 248
string, terminator 37
string() 302, 375, 377
strlen() 157, 580
strncat() 157
strncmp() 157
strncpy() 157
strrchr() 158
struct 180
structures 21, 173, 174, 247
STSWITCH 309
SUB_m_ 273, 277
SUB12 263, 273, 277
SUB1n 273, 277
SUBbpn 273, 277
subdirectory 189
subexpression 327
subprogram 261
subroutine 22, 70, 71
subscript 57, 91, 97, 97, 171, 172, 178, 220, 249, 274, 571
subscript, constant zero 171
subscript, expression 57
subscript, negative 57, 61
subscript, scaled 60

subscript, zero 220
subscripted reference 172
subscripting 49, 80
subtraction 46, 53, 263
SUBwpn 273, 277
sum 385, 387, 393
swactive 325-6
SWAP12 263, 271, 277, 399-400
SWAP1s 271, 277, 371
swdefault 323, 324
swend 252
SWITCH 271, 277
switch 181
switch, body 323-5
switch, command-line 186, 293
switch, evaluation routine 209, 212-14
switch, null 294
switch, statement 30, 113, 116, 240, 241, 251, 257, 258, 310, 311, 313, 323, 324, 325, 326
switch, statement, when to use 171
switch, table 251, 576, 577
switch, table of address/value pairs 213
switch, undefined 294
swnext 247, 252, 325
SWSIZ 577
SWTABSZ 247, 577
swv 385, 388, 393
symbol 28, 572, 573
symbol table 174, 224, 252, 253, 254, 260, 261, 279, 287, 293, 300, 305, 307, 314, 315, 317, 325, 327, 328, 332, 333, 340, 351, 352, 357, 359, 361, 362, 364, 365, 366, 367, 368, 369, 370, 372, 373, 378
symname() 284, 287, 288, 291, 300, 304, 315, 372
SYMTBSZ 247, 572, 573
sz 314

T

tab 19, 27, 37, 378
terminate program execution 162
test() 318, 319, 332, 341-2, 352, 353, 379, 421
TF (trap flag) 435
toascii() 159
token 19, 20, 27, 28, 248, 283, 297, 299, 574
tolower() 159
topop 385, 388, 389, 392, 393
toseg() 259, 261
toupper() 159
trailer() 260, 261, 296, 305
trap flag 435

two's complement 34, 45, 47, 92, 93, 164, 387, 393
TYPE 253, 305, 306, 412, 420
type field 254
typedef 181

U

UCHR 254, 299, 310, 403
UINT 254, 299, 310, 340, 351, 355, 374, 375, 377, 403
ULONG 419
unary operators 101, 230, 333, 337, 354, 360
uncooked data 139
undeclared identifiers 174
undeclared name 42, 73, 94
underscore character 28, 40, 216, 218, 267, 273, 278, 279, 286–89
ungetc() 143, 151, 152, 153
union 21, 173, 181
UNIX 426
UNIX shell 186
unlink() 152, 557
Unresolved Externals 193
unsigned 44, 45, 46, 47, 55, 62, 75, 92, 93, 94, 181, 298, 299, 306, 309, 310, 314, 340, 343, 347, 349, 354, 355, 364, 372, 374, 403, 409, 410, 422
unsigned binary 34
unsigned char 299, 300, 306, 310, 314, 364, 403
unsigned char * 364
unsigned characters 580
unsigned comparison 101, 171
unsigned constants 580
unsigned decimal 155, 156
unsigned int 299, 306, 310, 314, 364, 403
unsigned int * 364
unsigned integers 92, 580
unsigned keyword 34
unsigned long 419
unsigned long int 419
unsigned operation 62
unsigned p-code 349
unsigned result 351
uppercase 27, 28, 159, 159, 160, 197
upward compatibility 173
USES 382
utoi() 155
utol() 423

V

VARIABLE 300, 301, 314, 315
variable 26, 28, 39, 89, 253, 329, 572
variable, address 41
variable, character 21, 39, 46, 47

variable, global 40
variable, initializing 40
variable, integer 21, 39
variable, local 25, 41
variable, name 29
variable, pointer 39
VERSION 293
void 77, 181, 304, 407, 581
volatile 181

W

while 181
while queue 257, 293, 320, 321, 325, 326, 404, 405, 572
while statement 19, 28, 118, 257, 258, 311, 572, 577
white space 19, 20, 27, 31, 147, 148, 153, 154, 159, 186, 283, 286, 288, 289, 299, 303, 397, 398, 399, 579
white() 286
wild card characters 140
wipeout 140
WORD 262
word aligned 166
word boundary 166, 167, 210
WORD_ 270, 276
WORDn 270, 276
WORDr0 270, 276
wq 247, 258
wq[] 321, 404
WQEXIT 321, 404
WQLINK 404
WQLOOP 321, 404
WQMAX 257, 404
wqptr 258, 404
WQSIZ 257, 404, 572
WQSP 404
WQTABSZ 247, 257, 404, 572
write functions 139, 140, 153
write() 145

X

XENIX 426
XOR 167
XOR12 271, 278, 348
xtoi() 155
xtol() 423

Y

YES 136

Z

ZAPS 382
zero 95, 101, 102, 103, 153, 171
zero flag 435
zero tests 171
zerojump() 342
ZF (zero flag) 435
ZINU 426

A Small C Compiler, 2nd Edition

A Small C Compiler, 2nd Edition is an excellent resource for all programmers who want to learn the fundamentals of C, the most popular, professional programming language. A subset of the full C language, Small C includes most of C's features, allowing you to learn C without having to deal with its more advanced features.

This self-study course thoroughly explains Small C's (and therefore C's) structure, syntax, and features, and provides you with hands-on experience using C. Author James E. Hendrix succinctly covers the theory of compiler operation and design, discussing Small C's compatibility with C, explaining how to modify the compiler to generate new versions of itself, and suggesting several development projects to assist new C developers in quickly enjoying the flexibility and power of C.

A fully functional Small C compiler, plus a run-time library, all of the source code, and an executable assembler are included on disk in MS/PC-DOS format. With this complete package you'll immediately be on your way to enjoying a dynamic language and learning the mysteries of compiler operation.

Topics include:

- A thorough explanation of the Small C language
- Discussions on using the Small C compiler as a program development tool
- A revealing look at the inner workings of the compiler
- Appendixes that survey the 80X86 processors, explain error messages, and more

James E. Hendrix is best known for his further development of Ron Cain's Small C compiler. He is currently a researcher for the National Center for Physical Acoustics in Oxford, Mississippi. Hendrix has worked in numerous facets of the computer industry, writing programs and developing systems in the areas of science, engineering, manufacturing, banking, and university administration.

Why this book is for you- See page 1

M&T BOOKS

M&T Books, 501 Galveston Drive, Redwood City, CA 94063

52995

9 781558 511248

ISBN 1-55851-124-5

>$29.95